AFRL-IF-RS-TR-2006-236
Final Technical Report
July 2006

REFLECTIVE SELF-REGENERATIVE SYSTEMS ARCHITECTURE STUDY

Georgia Institute of Technology

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. U924

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

REPORT DOCUMENTATION PAGE	*Form Approved* **OMB No. 0704-0188**

1. REPORT DATE *(DD-MM-YYYY)* JULY 2006	**2. REPORT TYPE** Final	**3. DATES COVERED** *(From - To)* Jul 05 – Dec 05

4. TITLE AND SUBTITLE REFLECTIVE SELF-REGENERATIVE SYSTEMS ARCHITECTURE STUDY	**5a. CONTRACT NUMBER** FA8750-05-1-0253
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER 62301E
6. AUTHOR(S) Calton Pu, Douglas Blough	**5d. PROJECT NUMBER** U924
	5e. TASK NUMBER SR
	5f. WORK UNIT NUMBER SP
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Georgia Institute of Technology, College of Computing, 801 Atlantic Drive, Atlanta Georgia 30332-0280	**8. PERFORMING ORGANIZATION REPORT NUMBER** N/A
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Defense Advanced Research Projects Agency, 3701 North Fairfax Drive, Arlington Virginia 22203-1714; AFRL/IFGA, 525 Brooks Road, Rome New York 13441-4505	**10. SPONSOR/MONITOR'S ACRONYM(S)**
	11. SPONSORING/MONITORING AGENCY REPORT NUMBER AFRL-IF-RS-TR-2006-236

12. DISTRIBUTION AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED. PA#06-495

13. SUPPLEMENTARY NOTES

14. ABSTRACT
In this study, we develop the Reflective Self-Regenerative Systems (RSRS) architecture in detail, describing the internal structure of each component and the mutual invocations among the components. The main goal of RSRS architecture is a conceptually simple, but functionally rich description of the four topic areas of the Self-Regenerative Systems (SRS) program as well as the interactions among them. Specifically, we use the concepts of reflection and feedback control in the SRS components and between components. The feedback control is part of monitor-learning-actuator (MLA) loop. By allowing recursive and mutual invocations of the SRS components and the presence of MLA loops at different component levels, we will be able to create a rich set of capabilities that demonstrate the power of an SRS application or system. This analysis of the current/planned results of SRS Phase I projects, as described by the RSRS architecture as the unifying framework is then applied to two military scenarios, TCT (Time Critical Targeting) and Hyper-D/Aegis.

15. SUBJECT TERMS
Self-Regenerative Systems, Self-Management, Replication, Regeneration, Diversity, Insider-Threat, Availability, Survivability, Reflection, Feedback Control.

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** Patrick Hurley
a. REPORT U	**b. ABSTRACT** U	**c. THIS PAGE** U	UL	82	**19b. TELEPONE NUMBER (*Include area code)***

Table of Contents

Table of Figures

List of SRS Phase I Projects Covered in This Study

- **Cognitive Projects (Cognitive Immunity and Regeneration)**

1. **Learn/Repair (Learning and Repair Techniques for Self-Healing Systems):** *MIT: Michael Ernst and Martin Rinard*
2. **Model-Based (Pervasive Self-Regeneration through Concurrent Model-Based Execution):** *MIT: Brian Williams, Gregory T. Sullivan*
3. **AWDRAT (Architectural Differencing, Wrappers, Diagnosis, Recovery, Adaptivity, and Trust Modeling):** *MIT: Howie Shrobe, Teknowledge: Bob Balzer*
4. **Cortex:** *Honeywell: David Musliner*

- **Diversity Projects (Biologically-Inspired Diversity)**

1. **Genesis:** *University of Virginia: J. C. Knight, J. W. Davidson, D. Evans, A. Nguyen-Tuong Carnegie Mellon University: C. Wang*
2. **Dawson (DAWSON – Synthetic Diversity for Intrusion Tolerance):** *Global Infotek, Inc.: James Just*

- **Redundancy Projects (Granular, Scalable Redundancy)**

1. **SAIIA (Scalability, Accountability and Instant Information Access for Network-Centric Warfare):** *Johns Hopkins University: Dr. Yair Amir, Purdue University Subcontract: Dr. Cristina Nita-Rotaru*
2. **IITSR (Increasing Intrusion Tolerance via Scalable Redundancy):** *CMU: M. K. Reiter G. R. Ganger P. Narasimhan A. Ailamaki C. Cranor*
3. **QuickSilver:** *Cornell: Profs K. Birman, P. Francis and J. Gehrke, Raytheon: Dr. L. DiPalma and P. Work*

- **Insider Projects (Reasoning About Insider Threats)**

1. **PMOP (Detecting and Preventing Misuse of Privilege):** *Teknowledge: Bob Balzer, MIT: Howie Shrobe*
2. **MIT-HDSM (Mitigating the Insider Threat using High-dimensional Search and Modeling):** *Telcordia: Eric Van Den Berg, Rutgers: Raj Rajagopalan*

1. Summary

The RSRS Architecture Study is an in-depth study of self-regenerative systems, with application to the projects funded by DARPA's SRS program. This study assumes reader familiarity with self-regenerative system concepts and it is not intended as a primer to such systems. Part 1 of this study analyzed the SRS Phase I projects using the RSRS architecture. It is attached as APPENDIX of this final report. Part 2 of the RSRS Architecture Study (this final report) considers how to move forward based on several factors. First, we analyze the current/planned results of SRS Phase I projects, as described by the RSRS architecture as the unifying framework. Second, we apply the analysis to two military scenarios, TCT (Time Critical Targeting) and Hyper-D/Aegis, based primarily on information gathered by the PI during a visit to Naval Surface War Center Dahlgren Division. Third, we inserted into the scenario analysis of the new requirements introduced by BAA 06-35 as appropriate.

Architecture. The RSRS Architecture is a conceptually simple, yet functionally rich description of the four topic areas of SRS program, as well as the interactions among them. RSRS uses the concepts of reflection and feedback control between SRS components and within each component to describe their internal structure and external interactions. The main feature of RSRS is the monitor-learning-actuator (MLA) loop, based on the concepts of reflection and feedback control. The MLA is present in each system layer and component that supports self-regeneration. By monitoring component and system behavior, systems and applications are able to detect attacks and recover through the regeneration of data and programs using diversity to counter attacks targeting the physical representation of application (or system) programs. The MLA abstraction also suggests a standard interface among the technology areas as well as supporting customized extensions for each project. In addition to the main MLA abstraction, RSRS also includes component tools for specific areas such as Diversity and Redundancy.

Interfaces. The RSRS description of component interactions can be captured by a set of standard interfaces. Such a standard interface defines the common functionality among the projects of each area. Furthermore, extensions to these interfaces can be designed to support unique features provided by individual projects. The RSRS interface design is based on events, which carry both significant state information (the monitoring part of MLA) and control commands (the actuator part of MLA). The state notification events have a high degree of composability between event types. The control events enable various feedback control mechanisms to be adopted, including distributed and centralized control. These interfaces will support a manageable integration of SRS Phase I projects and provide the SRS facilities to other applications and demonstration projects that need self-regenerative capabilities. A high-level specification of the general RSRS interface is given in Part 2 of this study. Development of detailed interfaces is premature at this stage, because most of the software products developed in Phase 1 are not designed as components with well-defined external interfaces.

Evaluation. In the previous report (Part 1), the RSRS architecture has been successfully applied to model the self-regenerative aspects of current SRS Phase I projects. These models can be used to compare the projects of the same area at a qualitative level. After reprising the Study Summary and Architecture Description from Part 1, we discuss the main useful components and technologies that have been developed in the Program to date as well as work remaining to be done. We then present a high-level description of an event-based interface that could be used to facilitate rapid integration of SRS Phase I components into a complete system. Instead of using the interfaces for a qualitative analysis of the SRS Phase I project deliverables (information that

we have yet to receive), we proceed with two military scenarios in which we look forward and attempt to use the lessons in this architectural study to apply the results of SRS Phase I projects to concrete application scenarios. We will discuss the research challenges as well as bases for confidence for SRS Phase II proposers. The first scenario is TCT (Time-Critical Targeting based on DCGS) and the second scenario is Hyper-D/Aegis/DD(X). For the Aegis scenario, we analyze the research challenges as well as basis for confidence of SRS Phase II projects. We also describe the applicability of concrete SRS Phase I project results (as proposed) to both scenarios.

In the rest of this report, the term SRS is used primarily to denote the SRS Phase I projects and tools, although much of the RSRS architecture and analysis apply to both SRS Phase I and SRS Phase II programs.

2. RSRS Architecture

This section is a copy of the same section in Part 1 of the RSRS Architecture Study (attached in the Appendix). It is included here to make this document self-contained.

2.1. RSRS Main Concepts

The main concepts used in RSRS (see Figure 1) are: (1) feedback control and reflection in the MLA loop for self-regeneration, and (2) recursive and mutual use of component tools such as Diversity and Redundancy. In Figure 1, we have liberally added recursive use of tools among the four key technology areas of SRS: (1) Biologically-Inspired Diversity, (2) Cognitive Immunity and Regeneration, (3) Granular, Scalable Redundancy, and (4) Reasoning About Insider Threats.[1] This level of interaction and integration may not be achieved at the end of Phase 1; however, an effective architecture for the SRS program must show the potential interfaces for integration among the tools and techniques being developed by these technology areas.

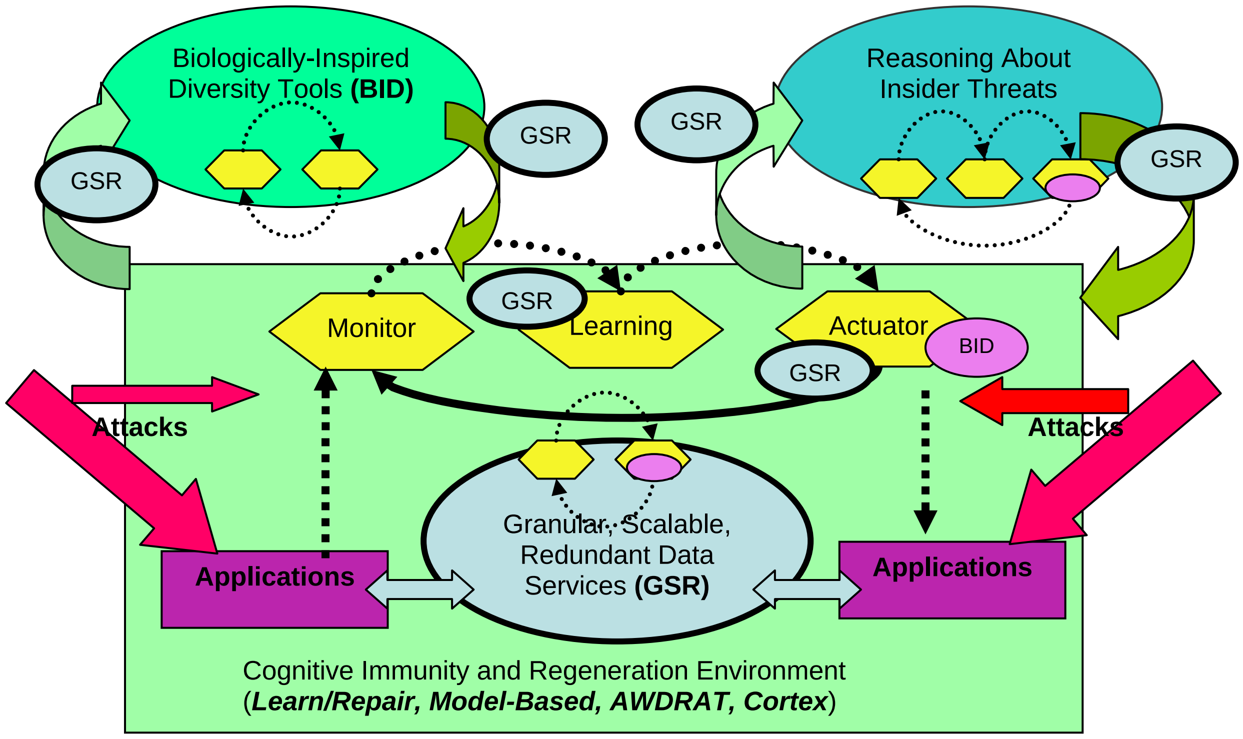

Biologically-Inspired Diversity Tools (BID)
Reasoning About Insider Threats
GSR
GSR
GSR
GSR
GSR
Monitor
GSR
Learning
Actuator
BID
GSR
Attacks
Attacks
Granular, Scalable, Redundant Data Services (GSR)
Applications
Applications
Cognitive Immunity and Regeneration Environment (Learn/Repair, Model-Based, AWDRAT, Cortex)

Figure 1 RSRS Functional Architecture

MLA Loop. The foundation of RSRS is the monitor-learning-actuator (MLA) loop, based on the concepts of feedback control in engineering systems and reflection in programming languages. The MLA loop consists of three components; (1) a monitor that reflectively observes system behavior (e.g., in applications running inside a Cognitive Immunity and Regeneration environment) or watches for anomalies in other parts of the system (e.g., insider threat monitors), (2) an actuator that takes self-regenerative action to recover from observed anomalies, and (3) an

[1] For brevity and where there is no ambiguity, we will refer to area (1) as "Diversity", area (2) as "Cognitive", area (3) as "Redundancy", and area (4) as "Insider".

optional learning component that records events observed by the monitor and actions taken by the actuator, and manages knowledge for better decision making in self-regeneration actions.

Role of MLA. The MLA loop can be found in all four technology areas of the SRS program. The MLA loop is represented in Figure 1 as yellow hexagons linked by connecting arrows. The loop is most prominent in the Cognitive[1] area, where four projects are building environments that support the monitoring, learning, and self-regeneration of application and system components to make them resilient against attacks. The MLA loop is also a major structural component of the two projects in the Insider[1] area, since such systems must observe the system behavior to detect and then counter insider attacks. The loop is more implicit in the two "service" technology areas. For example, the three projects in the Redundancy[1] area provide redundant data delivery services to improve the availability and reliability of the entire system. It is natural that these data services utilize MLA tools reflectively to improve their own availability and reliability. Similarly, the two projects in the Diversity[1] area, which create program and data generation tools, can use MLA to avoid monoculture vulnerabilities in their own tools.

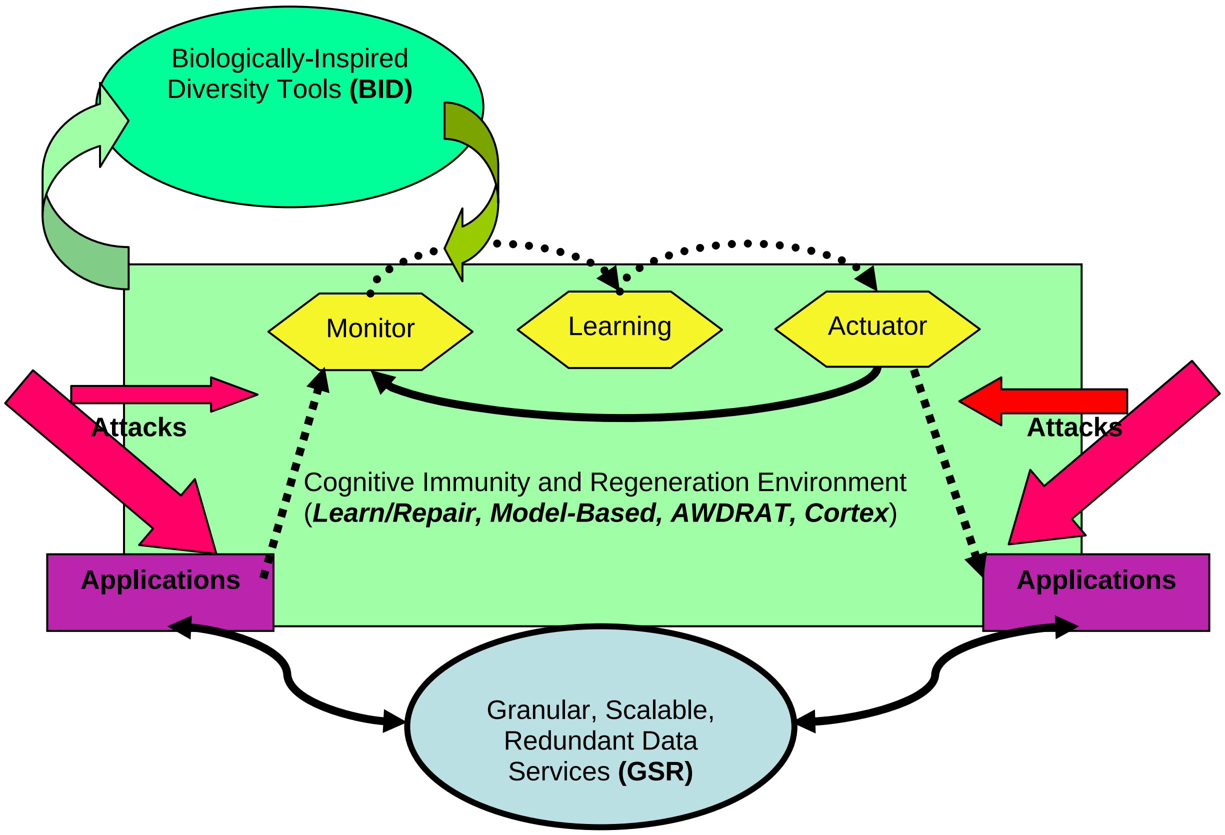
The MLA loop primarily captures the self-regenerative aspect of SRS projects. In the Diversity and Redundancy areas, the special capabilities provided by the projects are not necessarily self-regenerative. When discussing the inherent (non-self-regenerative) functionalities of components in these areas, we refer to them as "tools".

Diversity Tools. In the Diversity area, the projects will provide tools to create variants of binary representations of programs. These variants should have sufficient differences among them so attacks that depend on specific bit-layout (e.g., typical buffer overflow attacks) would not work on all variants. The variants are used by self-regenerative programs (e.g., in the Cognitive area), but the tools that generate the variants may be passive. The degree of difference among the variants (and the associated resistance to attacks) is outside the scope of the MLA loop model.

Redundancy Tools. Similar to the Diversity Tools, the Redundancy area provides tools for data replication, object replication, and reliable communication, which have the capability to resist certain types of attacks through redundancy. The performance and scalability of the specific redundancy mechanisms, while important to the SRS program, is outside the scope of the MLA loop model.

2.2. RSRS Analysis of Cognitive Projects

Cognitive MLA. In the Cognitive Area, the self-regenerative healing process can be described using the MLA loop. The three phases of self-regeneration (monitoring, learning, and regeneration) are outlined in this section and illustrated in Figure 2.



Biologically-Inspired Diversity Tools **(BID)**
Monitor
Learning
Actuator
Attacks
Attacks
Cognitive Immunity and Regeneration Environment (*Learn/Repair, Model-Based, AWDRAT, Cortex*)
Applications
Applications
Granular, Scalable, Redundant Data Services **(GSR)**

Figure 2 RSRS Architecture for the Cognitive Area

Monitoring. The first part of the Cognitive MLA is a monitoring service, which will monitor events at several time granularities and at different levels of system services. This approach comes from the observation that system failures and malicious attacks may occur through events at all time scales and system levels. Consequently, it is necessary for an SRS system to monitor the vital signs and anomalous events at several appropriate time scales and levels of abstraction. At each level of time scale and abstraction (e.g., hardware, kernel, middleware, and application process), the monitoring service will observe meaningful system states, compare them to acceptable states (using potentially different models and techniques), and generate events if significant state changes are detected.

Learning-Based Diagnosis. The second part of the Cognitive MLA is a learning-based diagnosis service, which may vary according to the time scale of events and their level of abstraction. For example, diagnosis at the hardware and kernel levels may be based on control systems. In contrast, diagnosis at the middleware and application levels may rely on model-based reasoning or data mining techniques. We use an abstract learning process to model the various learning-based diagnosis services. Abstractly, the events observed by the monitoring service are stored in an Events Database (ED), and the diagnosis process is a comparison of patterns in the ED with patterns previous learned and stored in a Known Events Database (KED). The KED stores both significant event patterns (problems) and an appropriate response and defense for those problems. In model-based systems, the KED captures the knowledge represented by the models. The learning part of MLA is represented by addition of knowledge into KED.

Regenerative Actuation. The third part of the Cognitive MLA is a regenerative actuator, which finds and carries out the recovery actions specified by the KED. For example, software updates may be available for a known virus using buffer overflow. In this case, KED will contain the recipe to apply the software update, or send an updated copy of system software to the affected node. If the problem is unknown (e.g., a new DoS or virus attack), then the Cognitive MLA enters a sub-loop to build and find an effective regenerative action using two high level services: a program diversity service provided by the Diversity Area and a data redundancy service provided by the Redundancy Area. The sub-loop will identify the extent of damage, create new system images to repair the damaged components, test them against the attack, find the variants that are effective against the attacks, store them in the KED, and distribute these variants in regenerative actions.

The sub-loop to search for remedies builds on the other SRS areas. For example, the new system image variants are created by program diversity tools from the Diversity Area. These variants may have been preventively generated beforehand, or dynamically generated at run-time. Similarly, trusted data and communications are provided by data redundancy services from the Redundancy Area. Each variant is then tested by creating an environment with the new variant and exposing it to the environmental conditions during the attack. If a new variant is shown to be resistant to the attack, it is entered into the KED and used to recover from the attack. The entire sub-loop may be online or offline, depending on the knowledge contained in the KED and the policies for recovery.

Cognitive Projects. Sophisticated software tools are being developed by the Cognitive area projects (***Learn/Repair, Model-Based, AWDRAT, Cortex***) that can be described by MLA loops. All of these projects have significant R&D efforts in monitoring, learning, and repairing of applications that run in their own environments. Although these efforts are currently isolated in their own projects, their components may be made available and interoperable through a standard interface based on the MLA abstraction. These tools can then be adopted by and integrated with other technology areas. Three of the four Cognitive area projects (*Model-Based, AWDRAT, Cortex*) employ model-based approaches where self regeneration is triggered by detection of deviations from a predictive model of the system. We note that the model-based approach is a specialization of the MLA approach, where in the monitoring component, system behavior is compared against model outputs, in the learning component, the system model is dynamically updated based on actual system outputs, and in the actuator component, adaptation of the system is performed based on model deviations.

2.3. RSRS Analysis of Diversity Projects

The Diversity area (*Genesis* and *Dawson* projects) develops code and data diversification tools that will be used by other areas and applications when attacks are detected. In Figure 1, the Diversity tools are represented as purple ovals, typically used by the actuator component of an MLA loop.

Figure 3 shows the RSRS architecture view of Diversity projects.

The main goal of Diversity projects is to develop Diversity Component Tools (algorithms and software) to achieve design diversity and data diversity goals. To measure the effectiveness of diversity algorithms, the evaluation steps can be modeled by an MLA-style loop.

- Monitoring: After the variants are created, their resistance to attacks is evaluated.
- Learning-Based Diagnosis: The winning variants are stored in a KED, while the losing variants are marked as such or discarded.
- Regenerative Actuation: The winning variants are then used to increase system robustness by replacing vulnerable components, possibly by a Cognitive component or system.

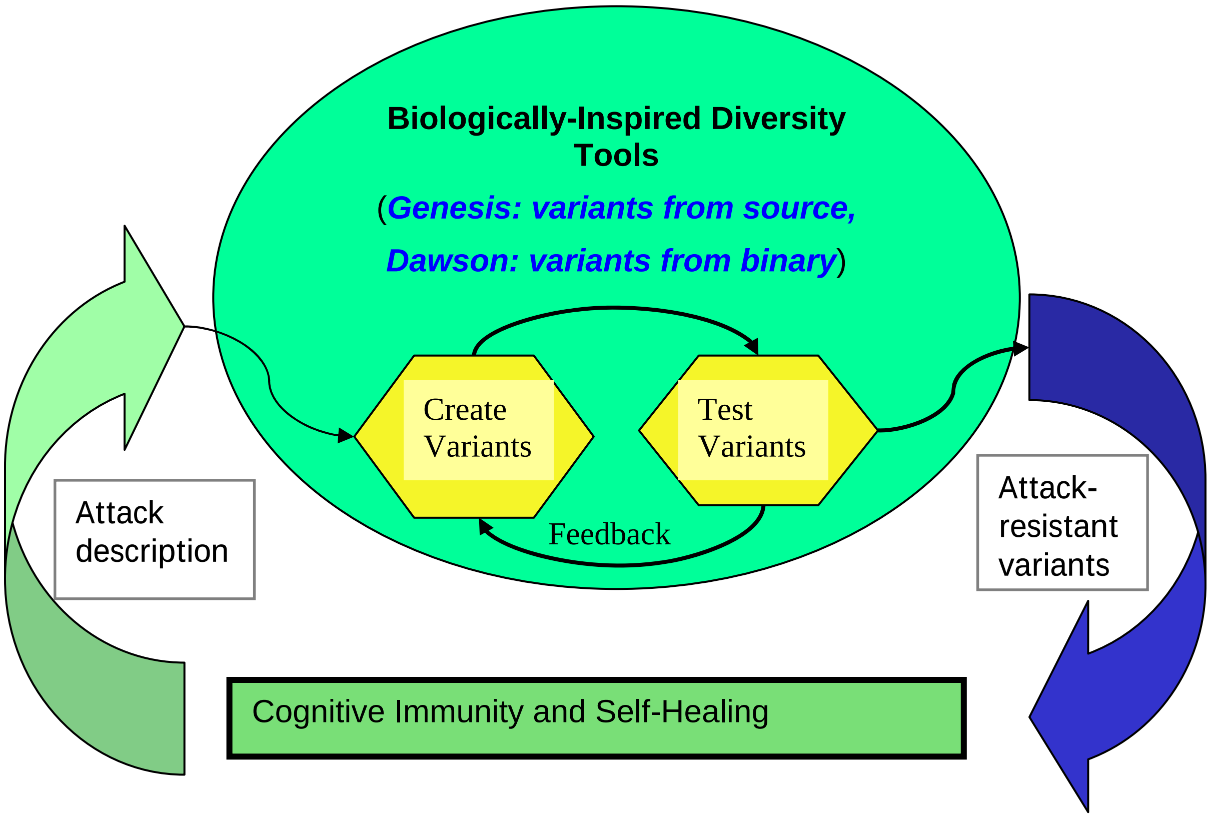
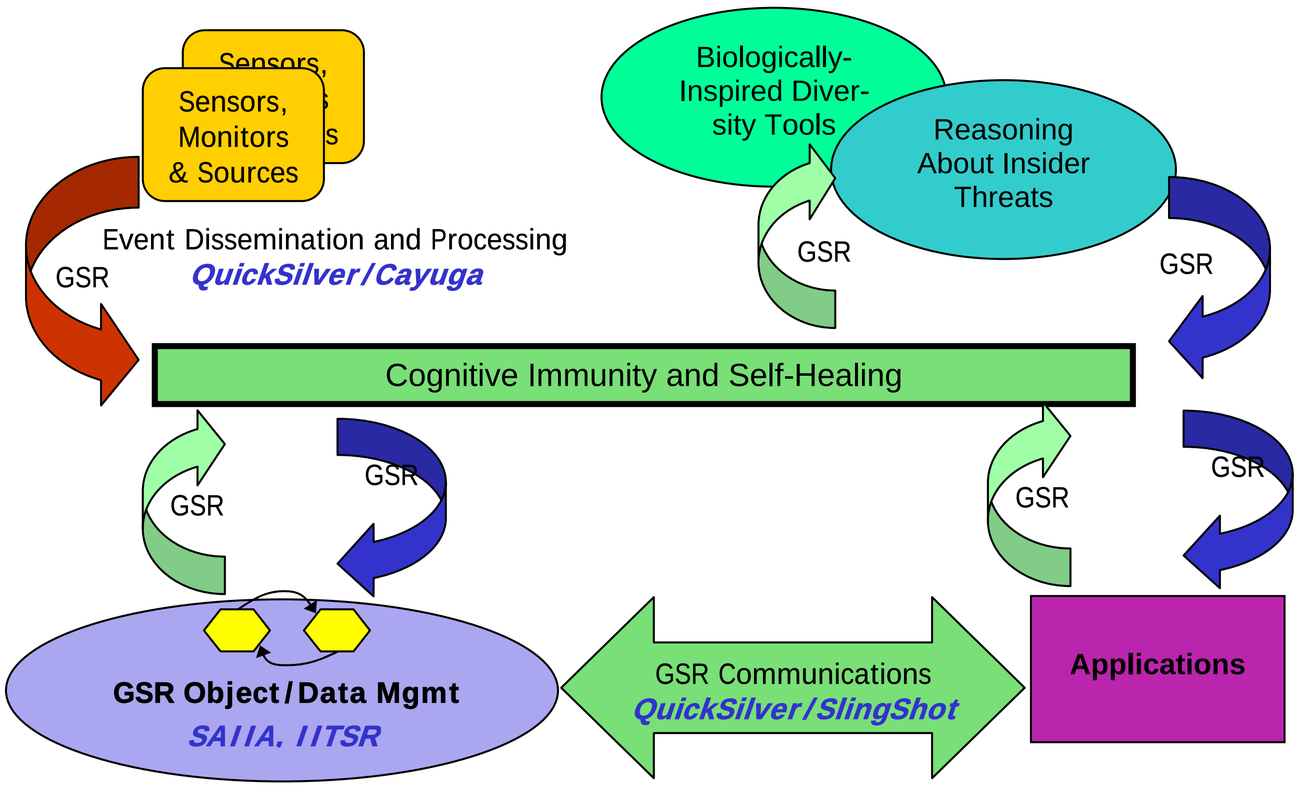

Biologically-Inspired Diversity Tools
(Genesis: variants from source,
Dawson: variants from binary)
Create Variants
Test Variants
Feedback
Attack description
Attack-resistant variants
Cognitive Immunity and Self-Healing

Figure 3 RSRS Architecture of Diversity Area

2.4. RSRS Analysis of Redundancy Projects

The Redundancy area projects develop highly-available services for communication, data storage, and computation for use by other areas and applications. For example, applications that run in a Cognitive Regeneration environment may have their communications and/or data flowing through channels created by the ***SAIIA***, ***IITSR***, or ***QuickSilver*** projects.

Figure 4 shows the RSRS architecture view of Redundancy projects.



Sensors, Monitors & Sources
Biologically-Inspired Diversity Tools
Reasoning About Insider Threats
Event Dissemination and Processing
QuickSilver/Cayuga
GSR
GSR
GSR
Cognitive Immunity and Self-Healing
GSR
GSR
GSR
GSR
GSR Object/Data Mgmt
SAIIA, IITSR
GSR Communications
QuickSilver/SlingShot
Applications

Figure 4 RSRS Architecture of Redundancy Area

2.5. RSRS Analysis of Insider Projects

The Insider technology area (*PMOP* and *MIT-HDSM* projects) consists of meta-level software tools that monitor system-level events and compare these events to a correct behavioral model or acceptable norm. Their MLA loops contain significant monitoring components (many system and application events may be relevant), learning components (dynamic adaptation may be required when under insider threats, e.g., temporary adoption of more conservative security rules), and actuator components (e.g., generating and switching to a different set of program modules that implement a more restrictive set of rules to counteract the suspected threat). The Insider area projects are represented as peach-colored ovals in Figure 1.

Figure 5 shows the RSRS architecture view of Insider projects.

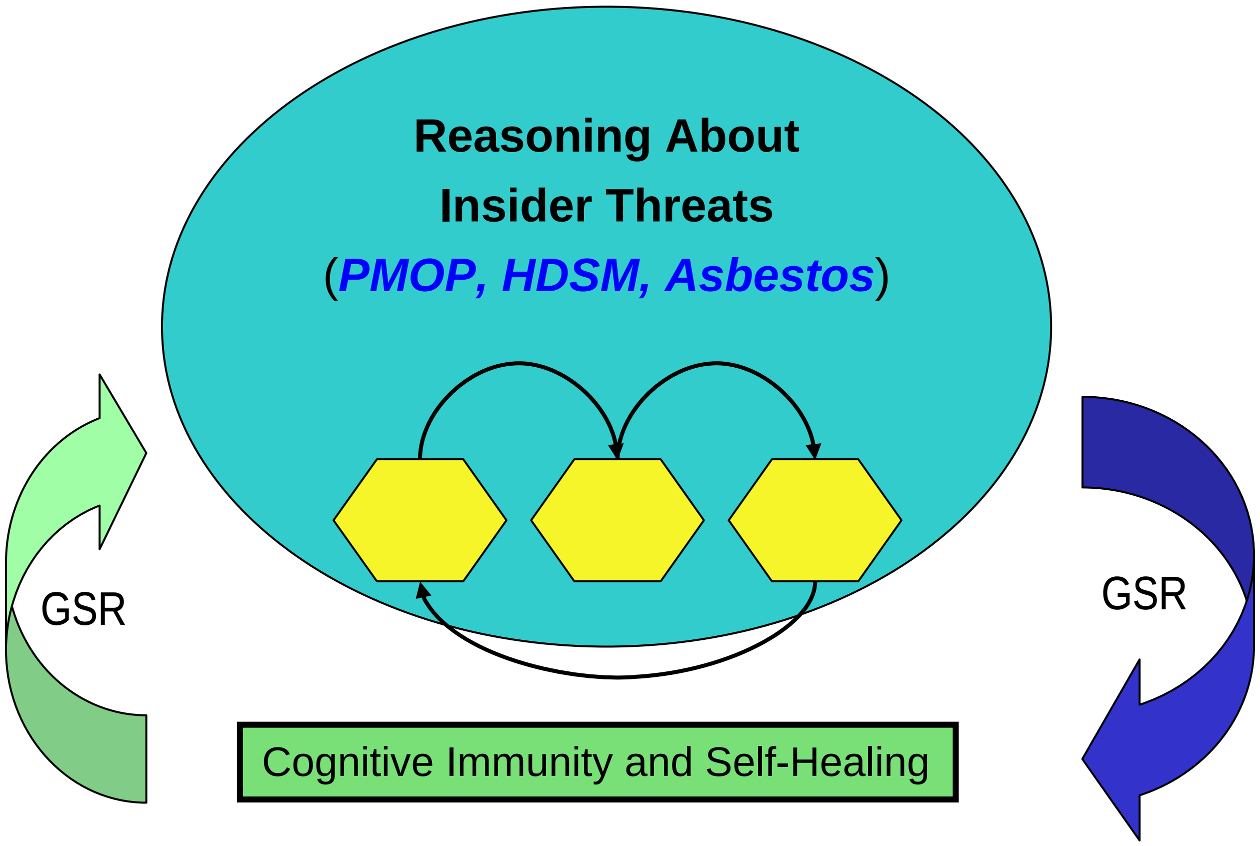


Figure 5 RSRS Architecture for the Insider Area

3. Summary Evaluation of SRS Program

In this section, we provide a summary evaluation of the current status of the SRS Program. Qualitatively, each project is compared to the core functionality of the topic area, and the unique features compared to each other. We point out specific tools and components developed by the projects that can be useful in building self-regenerative systems. We also point out where further work remains to be done before a comprehensive system can be built and demonstrated.

Finally, we focus primarily on the capabilities and goals of current SRS projects in this proposal. However, it is likely that the current projects will not fill all the components of the RSRS architecture. Furthermore, RSRS points the way to potential new building blocks, e.g., the capability of sites to dynamically generate and distribute new (diverse) code to handle newly discovered vulnerabilities. This may involve techniques not addressed by current projects, such as the use of mobile code generation and management techniques. This is a potentially interesting extension, whether using new techniques or existing mobile code platforms such as Java. It is one of the objectives of this study to develop an architecture that enables additional SRS capabilities to be developed and integrated seamlessly with the suite of capabilities being explored by current projects.

3.1. Summary Evaluation of Cognitive Projects

Summary of Learn/Repair Evaluation. The Learn/Repair project has a clear self-regenerative nature. The scope of the Daikon tool is within a program, concerning the regeneration of decayed data structures. This technique can be applied by application programs to become more robust against internal data structure corruption. ***Learn/Repair is the only project working on self-regenerative techniques applied inside a program.***

Summary of Model-Based Execution Evaluation. The Model-Based Execution architecture contains a monitor that detects faults during execution by comparing the execution results against the predictions of a model. Their Learning-Based Diagnosis module can define novel recovery actions for novel faults. The system method dispatch module can cause methods to be self-regenerated through redundancy and/or self-optimized through decision-theoretic approaches. ***The Model-Based Execution method may be applied to any application for which appropriate models can be created.***

Summary of AWDRAT Evaluation. The AWDRAT system builds an architectural model of each method executed. During the method execution, the model is evaluated concurrently and a monitor called Architectural Differencer compares the model against the method execution results. If the Differencer finds discrepancies, the learning-based diagnosis module updates the trust model for future reference. If damage is detected after differences are found, the regenerative action restores necessary data to a consistent state. ***The AWDRAT tools can be applied to any application for which appropriate models can be defined.***

Summary of Cortex Evaluation. The Cortex project uses “Taster Databases” to find and filter out dangerous queries. If bad queries cause damage to a Taster Database, that query is not forwarded to the Master Database for actual processing. The Cortex system learns to distinguish bad queries from normal queries, and reconfigures the system to switch off damaged databases as well as regenerating new replacements. The Cortex system can be used by a self-healing monitor to protect Master Databases by managing the Taster Databases on behalf of the entire system. ***Cortex can generate self-regenerative SQL databases that could serve as components within larger self-regenerative systems.***

Comparison of Cognitive Projects (Figure 6**).** At the program level, the Learn/Repair project is structurally different from the other three projects. The AWDRAT and Model-Based Executive projects are similar in structure and in their model-based approach. They differ in the application area chosen for the demonstration of the technology. The Cortex project is structurally similar to AWDRAT and Model-Based Execution, but focuses on the SQL query as the application area.

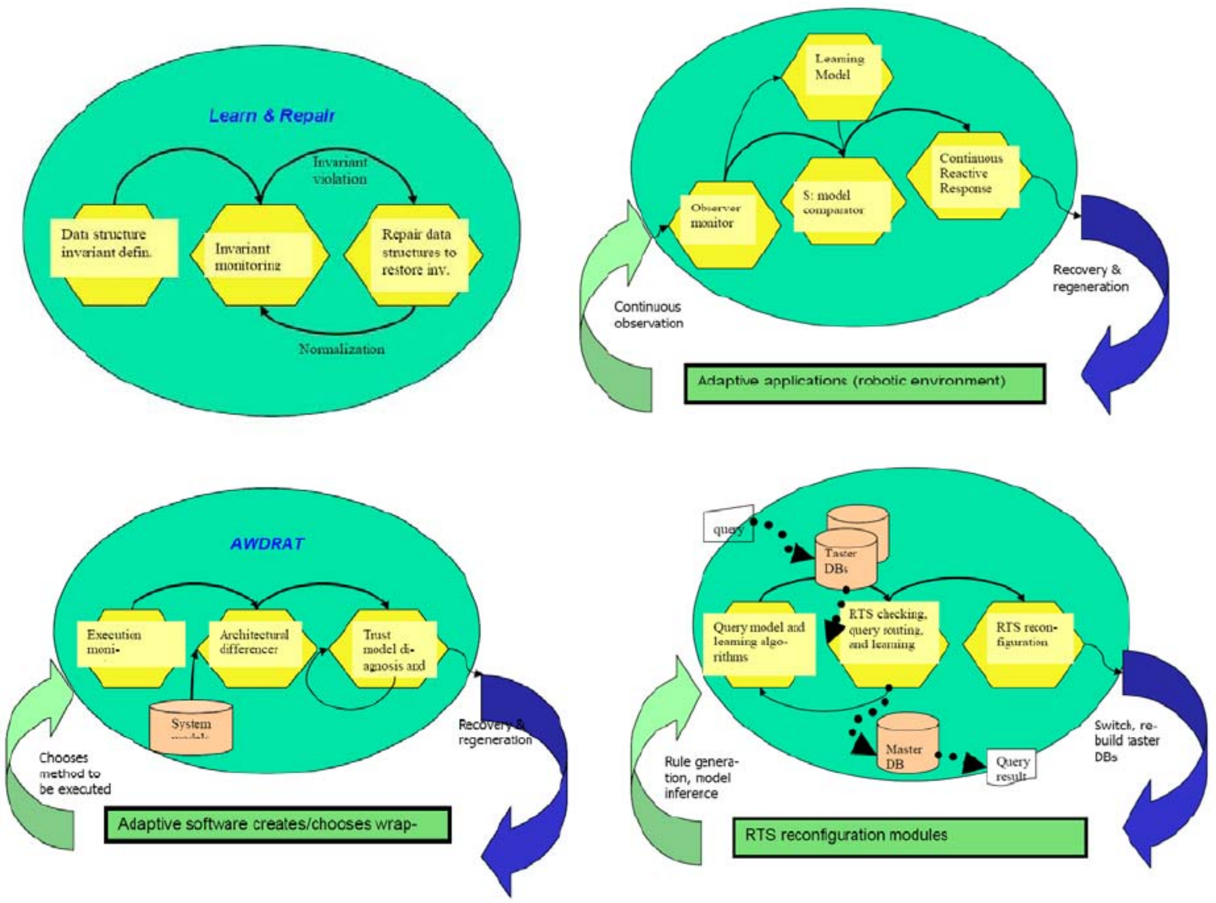

Learn & Repair
Invariant violation
Data structure invariant defin.
Invariant monitoring
Repair data structures to restore inv.
Normalization
Learning Model
Continuous Reactive Response
Observer monitor
S: model comparator
Continuous observation
Recovery & regeneration
Adaptive applications (robotic environment)
AWDRAT
Execution moni-
Architectural differencer
Trust model di-agnosis and
System
Chooses method to be executed
Recovery & regeneration
Adaptive software creates/chooses wrap-
query
Faster DBs
Query model and learning algo-rithms
RTS checking, query routing, and learning
RTS recon-figuration
Master DB
Query result
Rule genera-tion, model inference
Switch, re-build faster DBs
RTS reconfiguration modules

Figure 6 RSRS Architectural Comparison of Cognitive Projects

3.2. Summary Evaluation of Diversity Projects

Summary of Genesis Evaluation. The Genesis project generates program variants using techniques such as Calling Sequence Diversity and Instruction Set Randomization. The program variants can be tested and shown to be resistant to specific attacks. Some of the variants may also be immune to new attacks, for example, due to Instruction Set Randomization, which is difficult to guess. ***The GENESIS tool may be used by a system-level self-regenerative actuator to replace vulnerable programs or components either before or after an attack has happened.***

Summary of DAWSON Evaluation. The DAWSON project generates program variants for the Windows environment using techniques such as variable location (stack/heap) randomization and address (DLL/IAT) randomization. The program variants can be tested and shown to be resistant to specific attacks. ***The DAWSON tool may be used by a system-level self-regenerative actuator to replace vulnerable programs or components, in a Windows environment, either before or after an attack has happened.***

Comparison of Diversity Projects (Figure 7**).** The two Diversity projects are structurally similar, but have different deliverables (different operating systems and system environment assumptions). In addition, Genesis generates variants at multiple phases: compilation, linking, loading and run time, while Dawson creates variants from binary code.

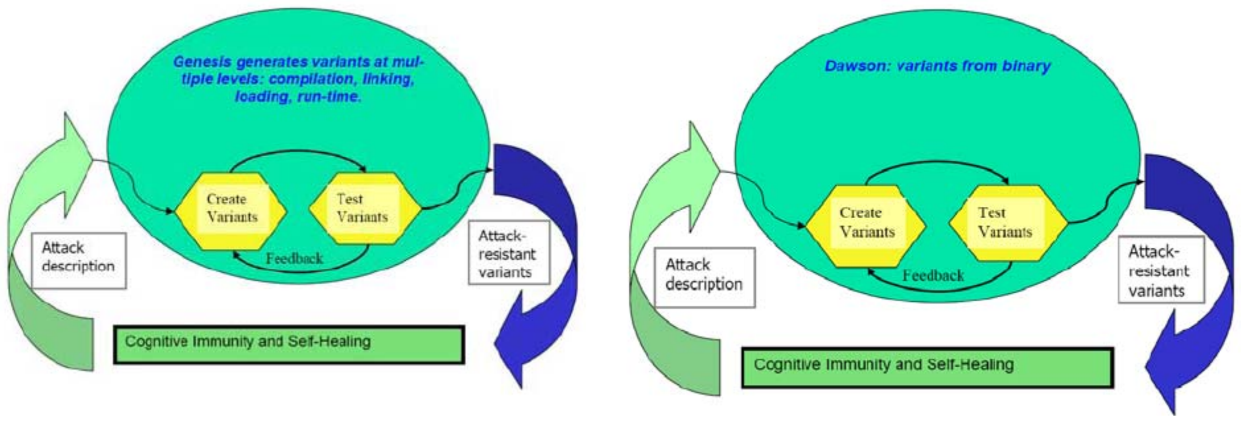

Genesis generates variants at multiple levels: compilation, linking, loading, run-time.
Create Variants
Test Variants
Feedback
Attack description
Attack-resistant variants
Cognitive Immunity and Self-Healing
Dawson: variants from binary
Create Variants
Test Variants
Feedback
Attack description
Attack-resistant variants
Cognitive Immunity and Self-Healing

Figure 7 RSRS Architectural Comparison of Diversity Projects

3.3. Summary Evaluation of Redundancy Projects

Summary of SAIIA Evaluation. The Steward wide-area object replication work is the most relevant to the SRS program. It provides a useful technology to support intrusion-tolerant systems based on replication that are deployed across wide-area network environments. Currently, the project has been primarily focused on achieving performance goals, as called for by the SRS program, rather than investigating self regeneration within a replicated environment. ***There is a significant future opportunity to enhance object replication mechanisms, such as those studied in the SAIIA project, by adding internal self regeneration so that they can not only support larger self-regenerative systems but can also provide inherently self-regenerative replicated objects.***

Summary of IITSR Evaluation. Currently, the IITSR project focuses primarily on providing supporting technologies for SRS. An unexplored but extremely promising extension would be to provide a self-contained self-regenerative data store, i.e. to incorporate aspects of self regeneration inside the data store itself. ***Techniques for diagnosis, recovery, reconfiguration, and adaptation of Byzantine-fault-tolerant data access technologies have been studied in several (non-SRS) projects. It should be possible to integrate these techniques with the types of technologies developed within IITSR to produce self-contained self-regenerative data stores.*** Self-contained self-regenerative components such as these could form building blocks from which larger self-regenerative systems could be constructed. Development of important self-regenerative components such as a general-purpose data store would therefore constitute an extremely valuable contribution of the SRS program.

Summary of QuickSilver Evaluation. The Quicksilver technologies fit well within the RSRS architecture. Important events, such as failures and intrusions that are detected at various levels by different monitors and sensors in the system, must be disseminated and processed using a system such as Cayuga. Applications communicate internally and externally using GSR communications mechanisms such as reliable multicast (SlingShot) and publish/subscribe (Quicksilver pub/sub). In order to support the RSRS architecture, all of the communication and event processing must be done in a scalable and reliable manner, using technologies such as those developed in the Quicksilver project. ***QuickSilver is the only SRS redundancy project focusing on scalable reliable communications services.*** The other two redundancy projects (SAIIA and IITSR) are focused on data and/or object replication mechanisms, rather than communication.

Comparison of Redundancy Projects. The three GSR projects are quite different in focus. SAIIA deals primarily with general object replication over wide-area networks. IITSR focuses on data replication with some consideration of flat objects. QuickSilver considers scalable and reliable communication. None of the GSR projects has considered internal self-regenerative aspects in detail; instead, they have focused on obtaining the best performance while providing functionalities of use to a larger self-regenerative system. Due to the redundant structures they employ, all of the projects have some capability to tolerate intrusions and attacks so long as they do not affect too many modules.

3.4. Summary Evaluation of Insider Projects

Summary of PMOP Evaluation. The PMOP project uses an operator behavior monitor to compare the expected actions (as defined by an Operational System Model) with the actual operations. If deviations are found, the Harm Assessment Module checks whether the extraordinary actions are dangerous. Dangerous actions go through Intent Assessment, which distinguishes malicious insider actions from operator errors. ***PMOP tools may be used to observe any applications for which a good set of models can be defined (Operational System Model, Harm Model, Intent Model).***

Summary of HDSM Evaluation. The HDSM project uses a large sensor network to collect behavioral information of operators. These data are stored in a network history repository, based on which a high-dimensional search engine will learn to distinguish proper actions from insider threats. An insider threat modeling and analysis tool builds models of insider knowledge acquisition that precedes attacks. A response engine performs impact analysis and synthesizes countermeasures that minimize potential damage. ***HDSM tools may be used to observe and detect insider threats provided appropriate sensors and models can be built and deployed for the operations being observed.***

Comparison of Insider Projects (Figure 8**).** The two Insider projects are structurally similar, with more emphasis on data mining in the HDSM project.

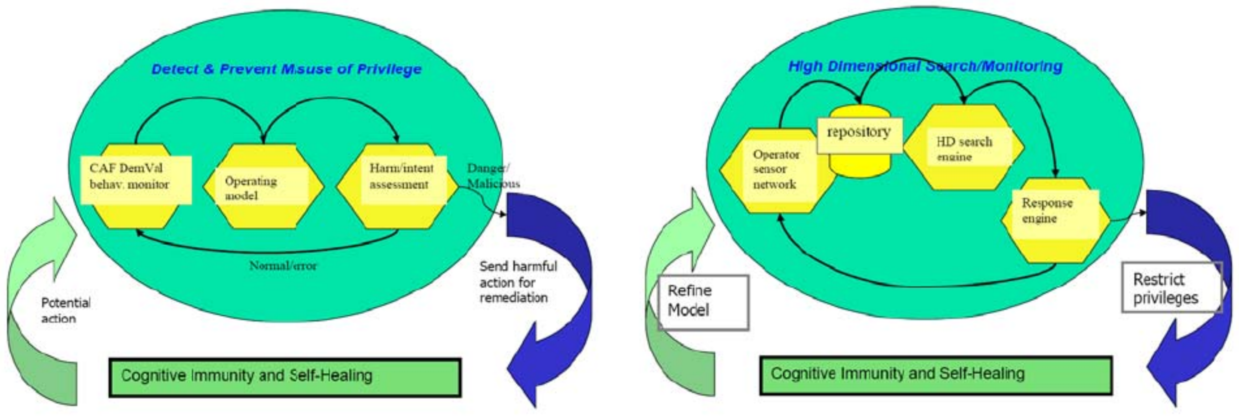

Detect & Prevent Misuse of Privilege
CAF DemVal behav. monitor
Operating model
Harm/intent assessment
Danger/ Malicious
Normal/error
Potential action
Send harmful action for remediation
Cognitive Immunity and Self-Healing
High Dimensional Search/Monitoring
repository
Operator sensor network
HD search engine
Response engine
Refine Model
Restrict privileges
Cognitive Immunity and Self-Healing

Figure 8 RSRS Architectural Comparison of Insider Projects

3.5. Summary of Potentially Useful Functionality from Current SRS Projects

3.5.1. Self-Regeneration Within Program Modules

Only one project: Learn/Repair of MIT, in the Cognitive area.

3.5.2. Diversity Projects

Creation of program variants with sufficient diversity to resist representation attacks (buffer overflow, etc).

3.5.3. Redundancy Projects

The following are useful functionalities created in the redundancy projects:

- Wide-area object replication protocols, threshold cryptography library (SAIIA)
- Data versioning, Byzantine protocols for read/write data, Byzantine protocols for query/update objects (IITSR)
- Scalable event processing, scalable reliable multicast, scalable publish/subscribe protocols (QuickSilver)

3.5.4. Cognitive Projects

Cognitive Projects (3 model-based)	AWDRAT	Model-Based Executive	Cortex
Application Model	OASIS DemVal (CAF)	Robots	MySQL
Monitoring	Model-based monitor	Model-based monitor of intent and procedures	scalable coherent state estimation
Diagnosis	**architectural differencing**, **alternative**	Compare observations with predictions	**Attack recognizer** (using probabilistic

	variant selection		reasoning algo.)
Learning	automated model update	Automated development of novel recovery methods for novel faults	Online learning – statistical and structural learning algorithms
Self-Regenerative Recovery	Recover data, select variant, and reinstall code	**Damage assessment**, contingency planning and execution	Plan and response, **Network Filter Generator**

3.5.5. Insider Projects

Insider Projects	PMOP	HDSM
Demonstration applications and scenarios	OASIS DEMVAL MAF/CAF	Human penetration followed by DDoS attacks, getting sensitive military information
Monitoring	behavior monitor using the operational system model	large scale sensor network
Diagnosis	**harm assessment** and **intent assessment**	high-dimensional search engine
Learning	Refinement of operational system model, harm model, and intent model	History repository and model/analysis tool
Self-Regenerative Recovery	Authorize or disallow actions that are not allowed by the operational system model	Response engine

3.6. Work Remaining

From the preceding material, it is clear that the SRS program has developed an impressive set of technologies that are necessary for development of self-regenerative systems. However, there are certain areas that still need to be addressed before the SRS technologies can be considered mature.

The projects that deal with the cognitive function for self-regeneration at the system level are all model-based. The model-based approach appears to be best suited for "single-application systems", e.g. embedded systems such as autonomous robots and UAVs that are self contained and focused on a single mission. For these types of systems, the development of a comprehensive model to describe the expected system behavior is a tractable problem and the model-based approach is a good one. However, applying the model-based approach to a complex system such as a military data center or a networked set of distributed resources that are cooperating on a multi-pronged mission appears to be an unsolved problem and may, in fact, be intractable.

Another approach is necessary to provide cognitive self-regeneration for large complex systems. The approach must be capable of reasoning about events that are generated at multiple time scales and from multiple system levels and it must be scalable to very large system sizes. We believe that the concept of reflection can be a guiding principle in the approach. A reflective system is one that continuously monitors its own state and adapts its operation accordingly. Reflection requires the ability to gather system state, the ability to reason about that state, and the ability to adapt system execution. System state can be gathered by monitoring events at various system levels and extracting relevant state information from them. This approach is shown in Figure 9, where events are gathered from monitors and sensors placed throughout the system and disseminated so that state information can be extracted. State information can be either extracted at the system level or it can be extracted locally and aggregated to form a system-level view. Adaptation can be achieved through mechanisms such as diversity using the types of tools developed in the DAWSON and Genesis projects, and also through resource managers that provides dynamic allocation of resources to tasks based on system mission priorities and current system state. Technology for this type of resource manager has been developed in the DARPA ARMS project (Adaptive Resource Management Systems). A system-level resource manager of this type is shown in Figure 9 and labeled as “Cognitive/Reflective System Manager”. Local resource managers can also make use of reflection, e.g. to add or remove replicas from a computation in order to either increase robustness or free up resources from a given task. The final piece that needs to be developed to make this approach viable for SRS is the ability to reason about system state. One view of this process is that it involves inferring high-level state, such as “the system is under attack by a worm”, from lower-level state such as “within the past ten minutes five sites have stopped responding to all requests” at the same time that “intrusion detection alerts throughout the system have increased by 25%”. In moving forward with the SRS Program, a focus on cognitive/reflective approaches of this type is necessary.

Another area that must be addressed in moving forward with SRS is development of components based on SRS technologies. With few exceptions, the current set of SRS projects have been developed as stand-alone systems and not as components, i.e. they have no well-defined interface by which other components can utilize them. This could cause some problems for an integration effort based on these technologies. Our initial goal was to try to standardize interfaces for different project areas. However, due to the lack of well-defined interfaces, we were not able to accomplish this. Instead, what we have done is to define an event-based interface, which is simple but also quite general. This should enable a diverse set of technologies to be “componentized” easily and integrated together quickly for demonstration of a complete self-regenerative system. The high-level specification of this event-based interface is given in the next section, along with a second level of detail for redundancy area projects as an example of how the interface would be used.

As mentioned previously, another opportunity for the SRS program to have impact would be to develop technology for some specific self-regenerative components, which could be used within larger systems. For example, in the redundancy area, self-regenerative data stores and self-regenerative objection replication components could be developed. This would provide a valuable contribution to the field because it would provide some initial building blocks that could become the basis for larger self-regenerative system projects both within DARPA’s purview and in the broader community.

4. RSRS Interfaces

One of the contributions of an architectural study is the capture of abstractions and definition of interfaces among the abstract components. In the RSRS architecture, multiple levels of MLA interfaces will be needed, as well as interfaces between the main components of the four technology areas. For the purposes of this study, we separate the RSRS interfaces into two parts: the *self-regenerative functionality* interface and the *inherent functionality* interface.

The self-regenerative functionality interface deals with interactions between components that are explicitly done for self regeneration purposes. For example, information about failures or attacks that are detected by one component should be disseminated to other components in a standard form, so that the other components can take regenerative action. Also, in some cases, it might be appropriate for components to be externally controlled, e.g. if there is a global resource manager controlling regeneration activities. This part of the interface allows components to be controlled in a parameterized fashion for regenerative purposes. It also allows components to specify regenerative actions to be taken by other components.

The inherent functionality interface applies primarily to components that are used as tools within an overall self-regenerative system. This part deals with the inherent functional interface of those components, i.e. the portion that is not directly related to self regeneration. For example, a data replication component might have data read/write and a set of metadata operations as its inherent interface. A scalable communication component might have a publish/subscribe interface or a set of multicast operations or some other interface that deals only with the communication aspects but *not* with self regeneration.

In the remainder of this section, we focus solely on the self-regenerative functionality interface, which we also refer to as the *RSRS-functional interface.* The inherent functional interfaces of the components are quite specific to individual tools and have little or no impact on the SRS architecture nor the self regeneration process. Standardizing the RSRS-functional interface across program areas and components will facilitate integration of the software produced by a diverse set of projects into a complete and cohesive SRS system.

4.1. Structural View of RSRS Architecture

Figure 9 depicts a structural view of the RSRS architecture applied to a single complex system, for example a military data center. In this view, there is a system-level cognitive component, instantiated by a cognitive/reflective system manager, which employs an MLA loop to monitor, learn, and regenerate at the system level. There is also an event disseminator component, which disseminates, throughout the system, events that are detected by monitors, sensors, and specialized detectors such as IDS components. There are scalable multicast mechanisms for communication among application components and mechanisms for gathering system status information and distributing regeneration commands. The MLA loops can be present within individual components and applications also. The monitoring components and specialized detectors can be viewed as a virtual sensor network, which generates data for the cognitive/reflective system manager to analyze. This opens the interesting possibility of using high-dimensional search techniques, which have been proposed in SRS for mining data from large-scale sensor networks, to analyze this data for the purpose of system-level event detection and diagnosis.

There is also an opportunity to include self-regenerative components that might be developed in SRS, e.g. a self-regenerative data store to maintain the knowledge base used by the cognitive/reflective system manager.

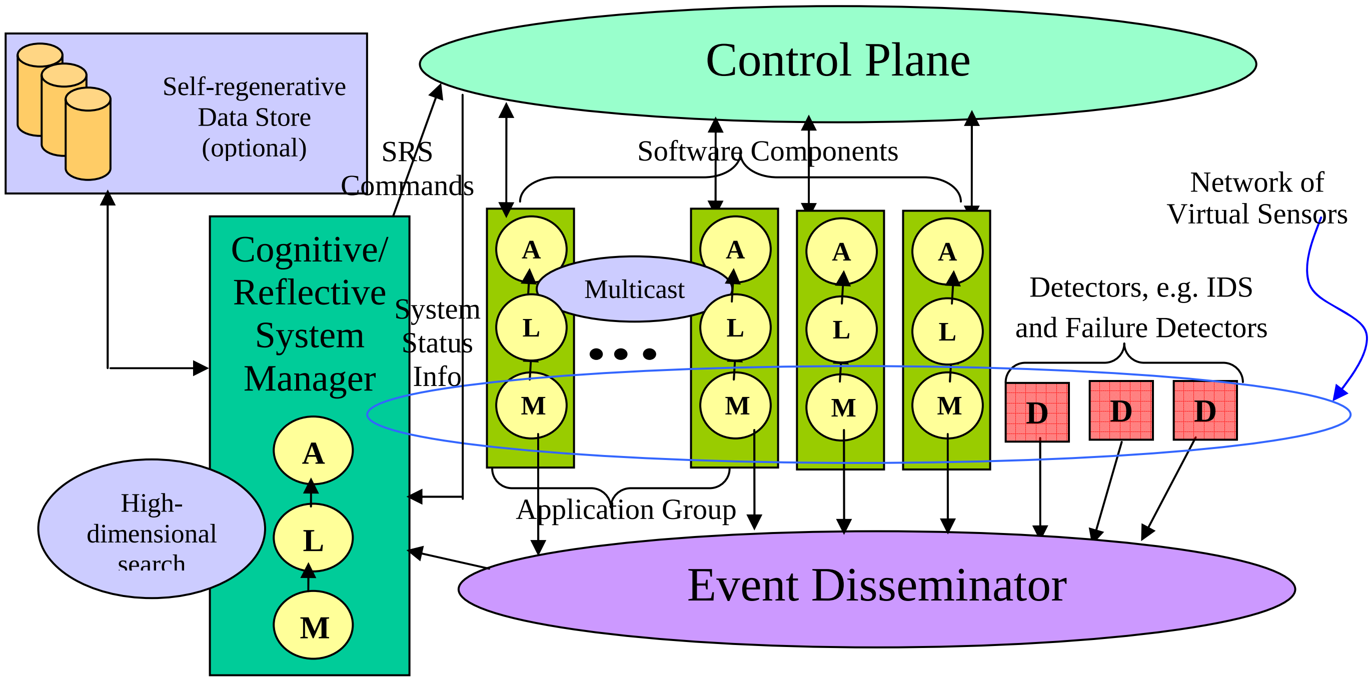

Control Plane
Self-regenerative Data Store (optional)
SRS Commands
Software Components
Network of Virtual Sensors
Cognitive/ Reflective System Manager
System Status Info
Multicast
Detectors, e.g. IDS and Failure Detectors
A
L
M
D
High-dimensional search
Application Group
Event Disseminator

Figure 9 Structural View of RSRS Architecture for Complex System

Figure 10 illustrates the RSRS architecture applied to a system of complex systems, e.g. the Distributed Common Ground System (DCGS), where each node is itself a complex system of the type depicted in Figure 9. In the RSRS system of systems architecture, each system performs local event dissemination and analysis. Any events that a system determines could be of interest to other systems are disseminated via a global event disseminator. These events will be processed by the cognitive/reflective system managers on other systems as appropriate. It is also possible that an entire node (or a portion of a node) is dedicated to analyzing events disseminated by the individual systems in order to detect global-scale events that can only be detected through identification of patterns of activity (event correlation) across the entire system of systems. An example of this type of global-scale event is a propagation-style attack, wherein similar suspicious events would be seen on different systems in a time-correlated fashion. As a concrete example, we could envision a military data center (or substantial processing resources from a military data center) being dedicated to analyzing events that are disseminated by individual systems within a DCGS environment.

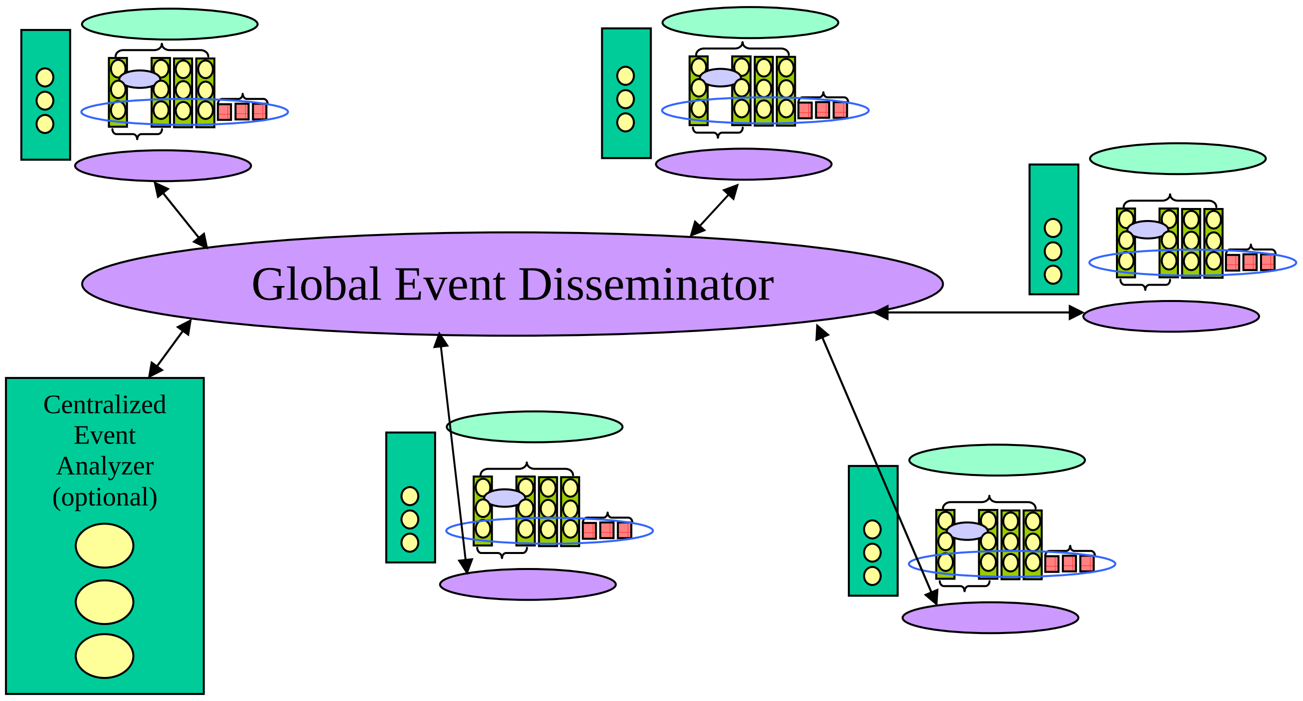

Global Event Disseminator
Centralized Event Analyzer (optional)

Figure 10 Structural View of RSRS Architecture for System of Systems

The structural views of the RSRS architecture shown in Figure 9 and Figure 10 provide a more concrete basis upon which to build interface specifications and to describe scenarios, such as the Time-Critical Targeting (TCT) scenario discussed later.

4.2. RSRS Interfaces: Overview

In keeping with the RSRS structural architecture, the RSRS-functional interfaces should be event-based, support a wide variety of event types and fields, and should also be extensible to allow applications to specify their own events. At a minimum, the interfaces should support common SRS-relevant events such as hardware/software component failures, recognized attacks, discovered software vulnerabilities, and control events. Web services and XML are obvious candidates for implementing interfaces that fit these criteria. In the following, we present skeletons for the RSRS-functional interfaces. It is our intention that these skeletons be used to guide the further development of these interfaces during Phase 2 of the program. Since most of the software products being developed in Phase 1 are currently not designed as components with well-defined external interfaces, development of detailed RSRS-functional interfaces is premature at this stage. Nevertheless, the skeleton interfaces and concrete usage examples described in the remainder of this section should provide a roadmap for detailed interface development in Phase 2. To the maximum degree possible, the different interfaces should be designed with common descriptions in order to standardize interactions between SRS components and with external entities as well. The interfaces should capture the core SRS functionality of each project area, and permit mutual and recursive invocation of different SRS components. Event-based interfaces allow information about failures, detected attacks, and control actions to be disseminated in a standard format (via GSR services) to relevant system components.

We include some basic interface data exchange formats here for illustration purposes. At a minimum, the information generated and processed by MLA will include (among other data fields):

- Event type field: [failure, attack, vulnerability, control]
- Failure fields: [failed component name, failure category, failure cause (if known), failure time, failure duration (if relevant)]
- Attack fields: [known/unknown, attack name (if known), attack category, attack time, attack duration, attack target]
- Vulnerability fields: [component name, module name, vulnerability category (if known)]
- Control fields: [control target, action type, action parameters]

These interface formats should be designed to be highly composable, e.g. a failure cause can be an attack, an attack target can be a vulnerability, etc. Control events allow components such as a global resource manager to be placed within a central cognitive component to dynamically distribute resource allocation decisions to the controllers of different system components. With this event-based approach, the basic interface operations are send and receive, which can be easily implemented with either publish/subscribe or message-passing systems. This approach should enable the diverse set of SRS projects to be adapted quickly in order to work together.

Event/Field Composition. The interfaces should allow fields and events to be composed in flexible ways. For example, a failure cause could be an attack (say a DoS attack that causes a server to crash or become unresponsive) and should allow the full specification of attack fields. Another example is that an attack target could be a known vulnerability and full information about the vulnerability should be included with the attack event. In addition, each event should be tagged with the ID of the site that generated it. At a higher level of abstraction, it is also desirable for a lower system layer to report causally related events to a higher layer in an aggregate form.

Extensibility of Events. We note that each project will have its own models and unique features that demonstrate its research contributions. The purpose of the standard interfaces is to provide a common ground of basic functionality that will be useful to the entire program. Each project's new capability or functionality can be added through customized extensions of these standard interfaces. Web service-style interfaces and XML-based data interchange will facilitate the incorporation of customized extensions. Furthermore, the standard and customized interfaces will support the recursive/mutual use of component capabilities from other projects in the SRS program. Standard interfaces will support alternative implementations of common functionality and customized interfaces will provide unique capabilities to other projects, whether they are in the same area or complementary areas.

Relation to RSRS Structural Architecture. As depicted in Figure 9, failure, attack, and vulnerability events are generated by various RSRS components and are distributed throughout the system by the Event Disseminator, which is a GSR communication component optimized for event dissemination and processing. Note that events can also be generated by non-SRS-specific components such as network-based or host-based intrusion detectors. The RSRS structural architecture and RSRS-functional interfaces allow such detectors to be integrated into the system. In the worst case, adaptors might be needed to translate outputs from generic detectors into standard RSRS format. The Cognitive/Reflective System Manager analyzes the event stream to determine what global regenerative actions need to be taken and it then distributes these actions to the appropriate software components via control events sent in the Control Plane. Individual software components can also query the event stream for relevant events and take their own local regenerative actions or request regeneration in other components.

In the following, we give examples of how the different project areas could make use of an event-based interface, and we provide a first level of detail in a possible interface for Redundancy area projects.

4.3. RSRS Interfaces and Cognitive Projects

Event Example. Consider an MLA loop created and maintained by a hypothetical Cognitive area project. The data generation and processing steps could be the following:

- Events are generated by plug-ins or monitor modules, and recorded by monitors.
- Events are processed and stored by learning components, if present.
- Events of broad interest (or aggregations of events) are disseminated via GSR services to other system components using the RSRS event-based interface
- Recognized attack patterns in the event stream trigger self-regenerative actions that are distributed between SRS components using control events from the RSRS event-based interface:
    - Diversity area tools generate appropriately diverse replicas with input parameters shown in the above example (e.g., failure and attack information).
    - New replicas are installed, initialized, data consistency recovered, and synchronized with other replicas.

Processing resumes in the application, which is now resistant to this kind of attack. In case of renewed attacks or failures, this step may be repeated with randomization to increase the chances of successful regeneration and recovery. This example illustrates how Cognitive area projects can make use of an event-based interface to interact with other components and perform successful regeneration while under attack.

4.4. RSRS Interfaces and Diversity Projects

Biodiversity components are essential pieces in the RSRS architecture. They provide one of the main methods of resisting, and recovering from, attacks. In terms of an event-based interface, biodiversity components can monitor the performance of their variants to learn of failures, attacks, and vulnerabilities. They can use this information internally to instantiate new variants with possibly different randomization parameters and can also disseminate this information to other components. These event reports could contain information such as which variants are vulnerable and which are resistant to a given attack. Similarly, they can receive reports from other components about failures and attacks and use the information to make local allocation decisions. In the case where a global controller/resource manager exists in the system, diversity components could also receive control events (commands) instructing them to take certain actions, e.g. create new variants or change randomization parameters.

4.5. RSRS Interfaces: Usage Examples for Redundancy Projects

The primary event types that we foresee being used by Redundancy area projects are failure, attack, and control. Data-replication-based diagnosis algorithms would make use of an event-based interface to notify other RSRS components of failure occurrences through the following fields:

- component name – failed servers can be identified by name (or IP address); distribution of this information to other RSRS components would allow them to optimize their data access patterns
- failure category – categories include, among others, crash fault, timing violation, data integrity failure, and Byzantine failure; this information could be used, for example, by the MLA loop within a Cognitive area project as part of system-wide event correlation to identify distributed attacks
- failure cause – identification of failure causes is highly dependent on system architecture and hardware/software configurations; examples of causes that could be determined by data-replication-based mechanisms are a DoS attack causing a server (or servers) to exhibit crash or timing failure symptoms, and local hardware/software problems causing multiple servers in the same subnet to experience simultaneous failures
- failure time – the time of the first observed failure symptom; this could be used, for example, by version-based recovery to determine which prior version to install

Intrusion and anomaly detection mechanisms, used in conjunction with Replication area projects, would make use of an event-based interface to notify other RSRS components of detected attacks via the following fields:

- known/unknown – recognized attacks, using e.g. signature-based techniques, would be classified as known; previously unseen anomalies would be classified as unknown
- name – names of known attacks would be distributed from Redundancy area components to other RSRS components; this could be used to trigger preset response scenarios by regenerative actuators in other components
- category – attack categories are virtually unlimited; general categories include confidentiality, integrity, and availability; specific categories include known attack types such as DoS, DDoS, man in the middle, spoofing, ARP poisoning, etc.
- time and duration – the time of the first observed symptoms and the duration if the symptoms have subsided could be used by higher-level mechanisms to analyze system-wide attack behavior
- target – if possible, names or IP addresses of nodes that are under attack could be disseminated

Failure and attack events could also be sent *to* Redundancy area components, for use by local controllers. Failures and attacks observed by other RSRS components could be used, for example, to dynamically modify replica configurations or adjust replication parameters. Local controllers could also request more resources from a resource manager component in response to failure or attack notification.

Control events could be used by Replication area projects to interact with components such as a global resource/configuration manager. Examples of control events that could be sent by a Replication component are requests to remove failed servers from active configurations, and requests to allocate additional servers either to increase resiliency or to replace failed servers. External controllers could send control events *to* Replication area components to force them to reconfigure, give up resources, or take other local actions that are necessary for overall system health.

4.6. RSRS Interfaces and Insider Projects

As an illustrative example, we will outline the external invocations of the Reasoning About Insider Threats topic area (abbreviated as "Insider"). An Insider component will contain an MLA loop to detect unusual activity that may indicate an attack and leading to reactive actions to such threats. When detecting an attack and damage, this MLA loop (the actuator) of the Insider component may send a failure notification event to a Cognitive Immunity and Regenerative Environment (abbreviated as "Cognitive") component causing it to regenerate a destroyed part of the system. In turn, the Cognitive component may send a control event to a Biologically-Inspired Diversity Tool instructing it to regenerate the damaged software modules. All the communications among the components may use a Granular, Scalable Redundant Data Service to increase the robustness of communications since the system is under attack. Similarly, each component may contain its own MLA loop to detect and recover from failures and attacks.

Note that, as depicted in the RSRS Structural Architecture, the monitoring elements within the MLA loops of different SRS components can be thought of as virtual sensors, or software sensors. Thus, the event streams from these monitoring elements and non-SRS-specific components such as intrusion detectors constitute a type of virtual sensor network. This facilitates the use of high-dimensional search techniques designed to mitigate insider threats using large-scale sensor networks, such as those being developed in Phase 1 by the HDSM project.

5. Discussion of SRS Deployment for the TCT Scenario

The following scenarios are illustrative examples that show potential applications of SRS techniques and tools in military situations. These discussions are in the context of an SRS Phase II program (Self-Regenerative Systems, Phase II).

5.1. TCT Illustrative Scenario

TCT Scenario. We will illustrate the military relevance of SRS research through a simplified scenario based on Time-Critical Targeting (TCT). TCT is an Air Force initiative building on fundamental capabilities such as CAOC (Combined Aerospace Operations Center), Predictive Battlespace Awareness, and Data Links. The goal of TCT is to find, fix, target, and engage valuable and perhaps mobile targets (e.g., mobile rocket launch platforms) in a single-digit number of minutes. A more concrete (and hypothetical) scenario is the following. Consider a tentative identification of a convoy moving away from an area with recent and current insurgent activities in Iraq desert. This is done by automatic target recognition software in a routine UAV surveillance video. High resolution satellite images are taken and the convoy's presence is confirmed. A TCT operation is initiated to stop and destroy the convoy before it can disperse or hide. This is a scenario typical of programs such as AMSTE (Affordable Moving Surface Target Engagement), where moving targets are discovered and tracked by long range GMTI (Ground Moving Target Indication) sensors and rapidly engaged with precision, stand-off weapons. For concreteness we will assign the task of identifying a potential target, fusing the information, and estimating its certainty primarily to DCGS (Distributed Common Ground/Surface Systems). This somewhat arbitrary assignment is due to the lack of detailed open literature description of programs such as TCT and AMSTE.

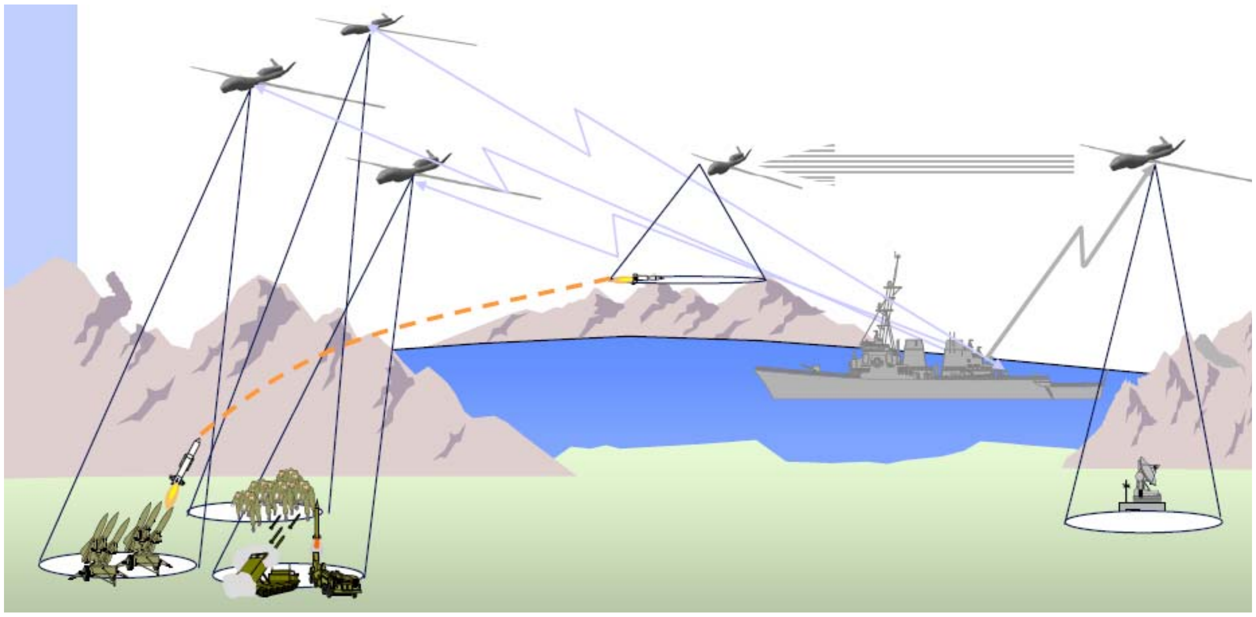


Figure 11 Command Post Scenario

Information Flow in TCT. DCGS is a family of systems capable of receiving, processing, exploiting, and disseminating intelligence in support of a Joint Force Commander. The AF DCGS is a worldwide distributed, network-centric, system-of-systems architecture designed to conduct collaborative intelligence operations by the Air Force. DCGS objectives include receiving imagery at ground and surface systems from national and tactical sensors and exchanging intelligence among the participants, for example, of an operation involving TCT. One of the major problems in such situations is the relatively long information flow and processing pipeline, which makes the TCT process vulnerable to attacks on any one of the system components. In our scenario, the convoy needs to be evaluated. First, it should be identified as friend or foe. Second, its value as a target should be established. Third, appropriate attack methods and means will be decided and resources allocated for the planned mission. Fourth, necessary information needs to flow up the command chain to obtain the authorization required for each mission. Fifth, an authorized mission should be carried out according to the plan, by distributing the sub-tasks to each unit responsible for carrying out the sub-task. Sixth, the completion status of each sub-task and of the entire mission such as damage assessment need to be sent back to the Command Post for the evaluation of the mission and of the entire battlefield.

The TCT scenario is summarized below in 12 steps.

1. TCT tasks are underway when a non-critical display application reports a data structure corruption event; the data structure is automatically repaired and the application continues; a few minutes later, another corruption is reported and repaired, although the application is forced to display at a lower resolution.
2. The RSRS cognitive/reflective component queries DCGS event streams for recent reports and notes that a larger-than-expected number of workstation crashes have occurred over the last 15 minute period.

3. The cognitive/reflective component then receives a report of errors from a replica, which is running a critical TCT task and is hosted on the same workstation as the display application.
4. A short time later, the workstation hosting the replica and display application crashes.
5. Critical applications use reconfigurable objects, so the system automatically starts a new replica on another workstation.
6. The RSRS high-dimensional search module is activated to analyze recent log and other event data within the Operations Center.
7. The search reveals unusual activity on the Operations Center gateway and a connection from the gateway to the crashed machine via a rarely-used port shortly before data corruption began.
8. The cognitive/reflective component also notes that the application using the port is on the list of applications that interact with the display application
9. The RSRS actuator takes the following actions:
   - It disseminates its analysis results (suspected application and port) to all other data/command/operations centers via DCGS.
   - It temporarily disconnects the Operations Center from DCGS and shuts down the gateway.
   - It reboots the failed workstation and disables the suspected application and port on all workstations.
10. Another data center, after seeing the Operations Center report, is able to capture and analyze the attack.
11. The attack info is then used by a bio-diversity generator to create a resistant variant of the targeted application, which it distributes to other centers via DCGS.
12. Once the TCT operation is completed, RSRS reconnects the Operations Center to DCGS, receives and installs the new variant on all machines, and reopens the closed ports.

5.2. Application of SRS Technology to TCT Scenario

This section can be retold from the TCT scenario point of view, along the information flow pipeline. This way, the applicable tools are applied for each stage of the pipeline. There will be some redundancy in this text organization, since some tools (e.g., diversity generators) can be applied to every stage of the information flow pipeline.

5.2.1. Self-Regeneration within Program Modules

[Functionality summarized in Section 3.5.1] The Daikon tool (Learn/Repair project) can be used to keep critical application and system programs running through the TCT information flow pipeline, e.g., the Automatic Target Recognition program used to identify the convoy, the friend-or-foe identification program, and so on.

5.2.2. Diversity Projects

[Functionality summarized in Section 3.5.2] Both variant generators (from the Genesis project and the DAWSON project) can provide variants of critical application and system programs that are resistant to known or new attacks. This can be done proactively, by mandating a certain level of diversity before attacks have happened, or reactively, in response to attacks such as the Slammer. These diversity tools can be applied to any or all of the programs along the TCT information flow pipeline.

5.2.3. Redundancy Projects

[Functionality summarized in Section 3.5.3] As a globally distributed system, DCGS needs robust communications mechanisms provided by QuickSilver (Cornell/Raytheon):

- Scalable event processing, scalable reliable multicast, scalable publish/subscribe protocols (QuickSilver), **robust scalable communications**

In addition, DCGS will also require reliable data storage services provided by SAIIA (JHU) and IITSR:

- Wide-area object replication protocols, threshold cryptography library (SAIIA), **corruption extent determiner**
- Data versioning, Byzantine protocols for read/write data, Byzantine protocols for query/update objects (IITSR), **scalable redundancy read/write distributed data storage**

5.2.4. Cognitive Projects

[Functionality summarized in Section 3.5.4] The three other projects in the Cognitive area are model-based or application-specific.

- AWDRAT will be demonstrated on OASIS DEMVAL (MAF/CAF GUI component).
- Model-Based Executive will be demonstrated on their robot environment.
- Cortex will be demonstrated on groups of MySQL databases.

Each project contains components that will monitor the application behavior, diagnose any problems (using learning techniques to anticipate new problems), generate solution plans for the problems, and carry out the solution using self-regenerative techniques.

In the TCT scenario, AWDRAT could be used to instrument some GUI (or other application) programs in communicating the Automated Target Recognition results to operators. If any deviations are detected, then those programs may have been compromised and they should be corrected. Similarly, Model-Based Executive could be used to instrument UAV or other sensor software, so their behavior can be observed and failures/deviations can be corrected automatically by software, complementing human operator guidance of the UAVs. This can be very useful when the number of sensors grows and exceeds the human monitoring capability. Finally, Cortex tools may be used to instrument backend databases for automated application tasks (e.g., for friend-or-foe recognition in a large coalition, mission planning with options, and workflow to obtain authorizations) to automate and mask the recovery from attacks or software failures of the database component.

5.2.5. Insider Projects

[Functionality summarized in Section 3.5.5] In a protected environment such as DCGS, insider attacks are a major concern. Both PMOP and HDSM will monitor operator behavior, compare it against a model, learn from the history, and decide whether to allow or disallow anomalous behavior.

PMOP has more explicit models of operational system, harm assessment, and intent assessment. These models distinguish (at an increasing level of sophistication) a malicious insider from operator errors. These models can be used to detect situations where a malicious insider operator may be trying to derail a TCT mission through abnormal interference, so such non-authorized interference can be stopped.

HDSM has a large sensor network and powerful high dimensional search engines that can learn and detect abnormal behaviors for a variety of applications and situations. Their demonstration includes human attackers penetrating machines to obtain sensitive military information or to launch distributed denial of service attacks. With sufficient training, these attacks can be detected at the initial stage (penetration) and stopped before that stage is successfully completed. The HDSM system can be demonstrated on a large distributed system such as DCGS to detect any problems covered by its sensor network.

5.2.6. Potentially Useful SRS Technology Not Addressed by TCT Scenario

The self-regeneration concepts and techniques being developed by the SRS program (and elsewhere) could be useful in some other ways for the TCT information flow pipeline.

- A system-wide cognitive SRS monitor that can replace failed components or entire subsystems.
- Broad definition of SRS: self-regeneration including/complementing reconfiguration techniques for fault-tolerance.

6. Discussion of SRS Deployment for the Aegis Scenario

6.1. Aegis Surface Combat System Scenario

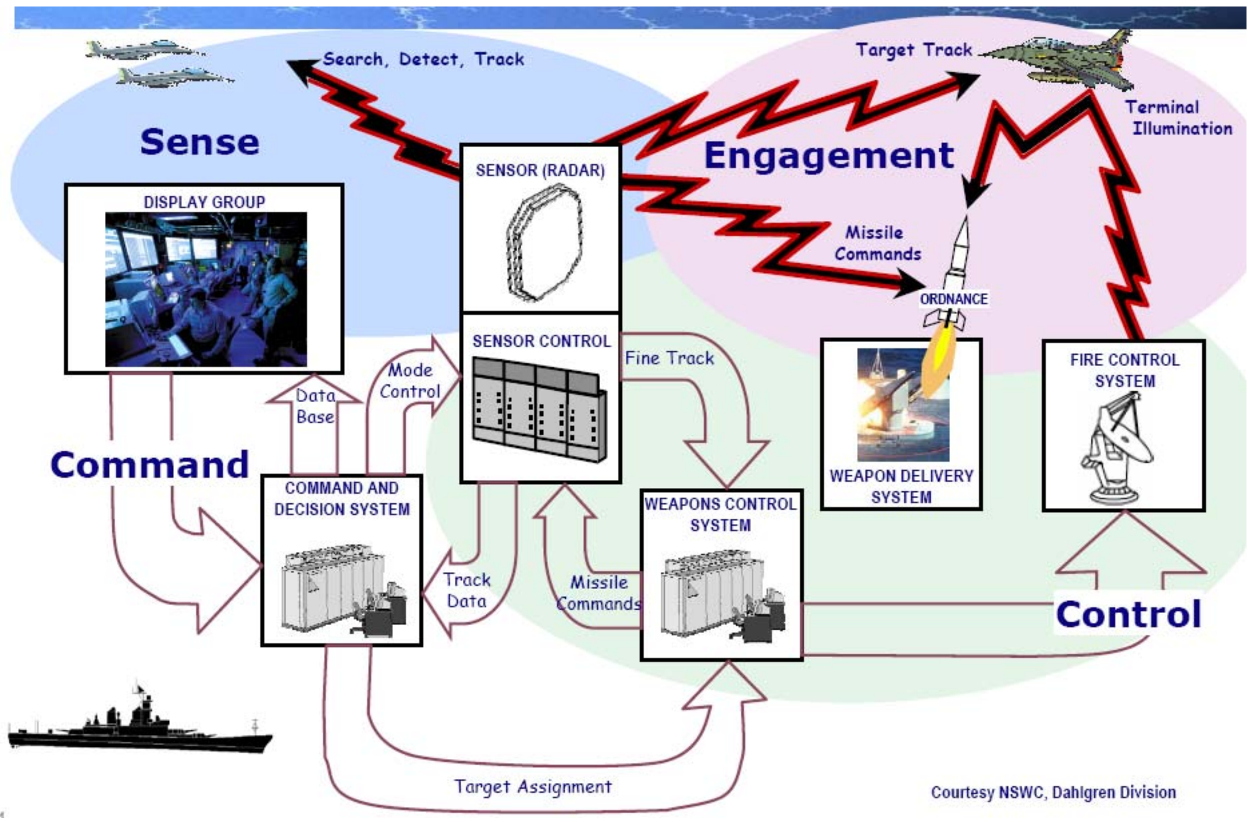


Figure 12 Typical Surface Combat System

Generally, weapons systems follow a design pattern comprised of three abstract fundamental functional areas: SENSE, CONTROL, and ENGAGE. These elements, briefly summarized, are:

- SENSE: Continually search for targets. A radar system that scans the area around a ship is an example in which the search function actively probes high-priority portions of the physical space. The search function may encompass multiple sensors (e.g., radar, lidar, infra red, magnetic resonance), in which case differing sensor outputs need to be combined. Data streams from sensors feed into a detection function that produces a stream of contact reports and predictions about contacts' likely future positions. Additionally, the SENSE functional area includes support for monitoring the engagement of weapons onto contacts.
- CONTROL: Consume contact reports and predictions from the SENSE functional area, perform identification on the contacts, evaluate their status if they are being engaged, prioritize them, predict the outcomes of engagement decisions, pair weapons and contacts, and configure the SENSE functions.
- ENGAGE: Receive contact and weapon information from the CONTROL functional area, and contact information from the SENSE functional area. Schedule the application

of weapons, compute fire control solution, prepare weapons, release weapons, and provide any necessary guidance for weapons in flight.

Figure 12 depicts a typical naval surface combat system such as the Aegis. An engagement process begins with the sensor finding a contact and delivering the "track data" to the "command and decision system" which updates a shared data base (the track file). An operator will typically oversee the "command and decision system" and manage by negation, which means observing the automated decisions the system makes and intervening only when necessary. "Mode control" messages return to the "sensor control" module to refine the sensor's operation to remove ambiguity and provide "fine track" information needed by the weapons control system (which itself also may have a human operator). The "display group" graphically portrays the data to the officers who are legally enabled to make engagement decisions. Such a decision then is forwarded to the "weapons control system" which both launches the weapon and monitors its operation in cooperation with the command and decision system.

The Aegis application is extremely sensitive to timing requirements. The radar and its sensor control module must provide contact records with low latency to the command and decision system, on the order of 1-10 ms. A contact usually comprises a few hundred bytes. As an example, the processing flow could comprise 47 messages per second with 516 bytes per message. The "sensor control", "command and decision system" and "weapons control system" have real time requirements in the millisecond range.

6.2. General Research Challenges of the Aegis Scenario

6.2.1. Related to millisecond real-time requirements

From the RSRS point of view, the MLA loop (monitor-learning-actuator) happens in each area of the Sense-Control-Engage pattern. In the Sense area, the monitoring is accompanied by learning, which identifies the critical attack patterns, and the actuator adjusts the sensors to sharpen their sensitivity to (and detection latency of) the critical attack patterns. In the Control area, contact reports and attack predictions from the Sense area are analyzed (the monitor part), their threat level assessed (with a learning part that reduces false positives and establishes priorities), and the weapons allocation decisions communicated to the Engage area (the actuator part). In the Engage area, both the contact and the weapons information are analyzed (the monitor part), the allocation among them optimized (through deterministic algorithms or through learning), and the decisions carried out (the actuator part).

The major research challenge for SRS Phase II presented by the Aegis scenario (referred to simply as "Aegis" in the rest of the section) is the real-time requirements at the millisecond range. The real-time requirements impose new challenges on SRS Phase II performers, since the SRS tools have not been developed to satisfy such stringent real-time constraints. Several significant modifications are required for SRS tools to be effective in Aegis. For example, the learning phase of MLA loop should be carried out in parallel, outside the critical path between the monitors and actuators. A second example is that the critical path between and including the monitors and actuators must be shrunk to the millisecond range. This means that any monitoring or actuating processes that need more than a few milliseconds will affect negatively the effectiveness of those SRS tools in Aegis.

The lack of time for an accurate diagnosis of the situation affects the Cognitive and Insider projects significantly, since their main goal is to determine what the problem is (through monitoring and learning). For example, one of the major challenges for model-based approaches (in both Cognitive and Insider areas) is the difficulty to generalize their results beyond the model, a serious problem with a prescribed scenario such as Aegis. With the real-time requirements, another major technical challenge for model-based approaches is to "compile" the model (even if limited in some sense) into very fast programs that can make good decisions within the millisecond range. (This is also related to the need to separate the learning process into an offline task.)

SRS Phase II Requirements. In BAA 06-35, new success criteria have been defined. The following are related to timing constraints:

- Diversity: new variants can be generated rapidly, i.e., in < 1 second.
- Cognitive: generate responses within 250 milliseconds of attacks.
- Redundancy and Insider: no explicit timing requirements, although compatibility with the above timing constraints is assumed.

6.2.2. Related to self-regeneration (diversity and redundancy)

In terms of self-regeneration, a major concrete consequence of modifications above is the requirement for offline pre-generation and pre-loading of useful variants by SRS Phase II tools. This requirement seems feasible for the Diversity and Redundancy projects (Sections 3.5.2 and 3.5.3), but more difficult for Cognitive, Insider, and Within-Program projects (Sections 3.5.1, 3.5.4, and 3.5.5). The pre-loading requirement creates the second concrete consequence, which is the lack of time to diagnose the attack and therefore the pre-loaded variant has to anticipate the kind of attack during its generation time. For new and unknown attacks, the lack of time for diagnosis will test the SRS Phase II tools' capabilities to pre-generate sufficiently useful variants that are different enough to resist unknown attacks before the diagnosis processes are completed.

Depending on the time required for switching on the pre-loaded variants, the variants can run as warm standby (lowest execution overhead, but slowest switch-on time), hot standby (medium execution overhead and medium switch-on time), or a concurrent execution with voting quorum (highest execution overhead, but fastest switch-on). Since warm standby seems unlikely to satisfy millisecond real-time requirements, we focus on hot standby and concurrent execution. For hot standby, the concurrent replay by the standby variants may or may not suffice to achieve data consistency maintenance among the variants, since explicit synchronization between the variants may be necessary unless we make the expensive assumption that complete serialized execution log records for the variants will be always in the exact same order. The "continuous" consistency requirement (or millisecond range lag) is non-trivial to satisfy, both across programs or within a program (e.g., the consistency check cycle of Daikon).

For concurrent execution of variants with voting on the results, new problems arise. For example, the execution speed of variants become a significant problem in voting, which will be dominated by the slowest voter. If variants have very different execution speeds, the entire system will be slowed down considerably. This is a problem for some current Diversity tools, which create variants that execute several times slower. Another serious research issue is the determination of which variants are executing correctly, since the majority may not always be right. Another challenge is to ensure that different variants do not introduce non-determinism that would violate the strong consistency requirements of software-replication-based voting techniques.

Beyond the main memory consistency problem, it is also unclear whether the current SRS replication tools will be sufficient when databases or files are being shared with other applications. For a demonstration program, it may suffice to use replicated data that is exclusively accessed by the demo application.

6.2.3. Basis for Confidence

The discussion in the previous subsection describes some of the research challenges due to the combination of real-time requirements and work required in the regeneration process during the Sense-Control-Engage process. There are several reasons that these requirements may be satisfied simultaneously. For example, much of the regeneration work can be done before the actual regeneration event, including the pre-generation of variants and pre-loading of variants into memory in the hot-standby and concurrent execution with voting scenarios.

The issue of real-time monitoring and diagnosis of attacks is more complex. Accurate diagnosis in real-time is not easy and errors have serious consequences. False negatives cause downtime and false positives cause self denial of service due to frequent switches. However, a model-based approach could conceivably build an attack model offline for demonstration purposes. This has been described by the model-based projects in the program. These pre-built models will be limited by the knowledge they contain, so they may or may not be able to diagnose unknown attacks in real-time. Although non-trivial, the problem is not hopeless. There are several possibilities that we enumerate here as feasible, although somewhat speculative, research challenges for SRS Phase II.

One approach to diagnosis is to use sufficient diversity in a shotgun approach, as an alternative to accurate diagnosis. If sufficiently diverse variants can be pre-generated and pre-loaded, then it is possible that new attacks may be countered by some of these variants. The difficulty in this shotgun approach is the need to generate “sufficiently diverse” variants, which is a serious and new research challenge.

Another approach to diagnosis is to use the potential results from other DARPA programs such as Application Communities, where one part of the community may be attacked and provide information for diagnosis of the attack. The diagnosis information may be used by other parts of community before the attack has reached them. If SRS Phase II tools are able to use the diagnosis information to generate attack-resistant variants and pre-load them as either hot-standby or concurrent execution voters, then it may be reasonable to assume that accurate real-time diagnosis will come from some other source, e.g., Application Communities program.

Another serious research for SRS Phase II tools is the consistency maintenance among the variants. There are some feasible approaches that should be explored and evaluated. For example, hot standby with completely serialized execution log and completely replicated data should be evaluated with respect to the synchronization time. Further, the costs of completely reliable logging and distribution of serialized log should also be evaluated. Similarly, concurrent execution voting variants also need to be evaluated with respect to synchronization time and consistency maintenance costs. It is also possible to assume asynchronous periodic checkpoint for another kind of hot standby in specific areas, e.g., during the ENGAGE phase.

6.2.4. Linkage with Other Programs

In addition to SRS (Phase 1), SRS Phase II projects may benefit from interacting with two other programs. First, the Application Communities program is developing techniques to monitor attacks and share the monitoring and diagnosis results quickly. This kind of diagnosis and prognosis information will be invaluable for SRS Phase II software tools in Aegis, which have little time of their own for accurate diagnosis.

Another program that may provide significant advantages to SRS Phase II projects is the Hyper-D software environment, which can be used for demonstration, integration, quantitative evaluation, and validation of SRS Phase II tools.

6.3. Application of SRS Techniques to the Aegis/Hyper-D Scenario

6.3.1. SRS Techniques within Program Modules

[Functionality summarized in Section 3.5.1] The Daikon tool (Learn/Repair project) can be used to keep critical application and system programs running through Aegis the Sense-Control-Engage information flow pipeline, e.g., the "sensor control", "command and decision system" and "weapons control system", and so on.

The millisecond-range real-time requirements will be a significant challenge to SRS tools, since the regeneration process needs to be done somehow within the severe time constraints. In case of the Daikon tool, the current cycle for detecting a problem and correcting it seems to be at the time frame of about a second. Since the detection and repair processes cannot be made offline, it is unclear how their approach can work effectively in Aegis.

Specifically for Daikon, there is an additional requirement for the Aegis scenario. The decision to engage depends on the assumption of a correct identification of the incoming target. When the Daikon tool repairs damaged data structures, it is not always guaranteed that the results are "correct" from the application point of view. Consequently, the Sense-Control-Engage pipeline needs to be marked with Daikon repair caveats, even when it's done within the time constraints.

6.3.2. SRS Techniques in Diversity Projects

[Functionality summarized in Section 3.5.2] Both variant generators (from the Genesis project and the DAWSON project) can provide variants of critical application and system programs that are resistant to known or new attacks. This can be done proactively, by mandating a certain level of diversity before attacks have happened, or reactively, in response to attacks such as the Slammer. These diversity tools can be applied to any or all of the programs along the Aegis Sense-Control-Engage information flow pipeline.

The BAA 06-35 mandates the generation of new variants to be less than a second. This requirement may be alleviated if the generation of new variants can be parallelized (to increase throughput) and carried out in anticipation of the attack (to decrease response time). For the Aegis scenario, due to the millisecond response time requirements, the pre-generation and preloading are required in any case.

The millisecond-range real-time requirements may be a significant challenge to SRS Phase I tools, since the regeneration process needs to be done somehow within the severe time constraints. The current setup of program variant replacement in both Genesis and DAWSON seem to require loading the new variant from the file system and re-initialization of the new variant. As an optimization to bypass the loading phase latency, the new variants will need to be preloaded into main memory, and activated as needed (a warm standby). As a further optimization, these new variants may be executed concurrently (a hot standby) and results voted, if full replicated and concurrent processing is required. This would be the case if the re-processing would cause the system to miss its original deadlines. Concurrent processing may also be desirable simply to speed up the Sense-Control-Engage pipeline, since an incoming target must be dealt with as soon as possible.

6.3.3. SRS Techniques in Redundancy Projects

[Functionality summarized in Section 3.5.3] As a tightly integrated networked system, the Aegis Sense-Control-Engage information pipeline can benefit from robust communications mechanisms provided by QuickSilver (Cornell/Raytheon):

- Scalable event processing, scalable reliable multicast, scalable publish/subscribe protocols (QuickSilver), **robust scalable communications**

In addition, the Sense-Control-Engage information pipeline can also benefit from reliable data storage services provided by SAIIA (JHU) and IITSR:

- Wide-area object replication protocols, threshold cryptography library (SAIIA), **corruption extent determiner**
- Data versioning, Byzantine protocols for read/write data, Byzantine protocols for query/update objects (IITSR), **scalable redundancy read/write distributed data storage**

6.3.4. SRS Techniques in Cognitive Projects

[Functionality summarized in Section 3.5.4] The three other projects in the Cognitive area are model-based or application-specific.

- AWDRAT will be demonstrated on OASIS DEMVAL (MAF/CAF GUI component).
- Model-Based Executive will be demonstrated on their robot environment.
- Cortex will be demonstrated on groups of MySQL databases.

Each project contains components that will monitor the application behavior, diagnose any problems (using learning techniques to anticipate new problems), generate solution plans for the problems, and carry out the solution using self-regenerative techniques. The main challenge for the model-based approaches is their ability to adapt the approach and create a model for the prescribed Aegis scenario. The potential contributions of these projects to SRS Phase II seem to depend on whether they are able to leverage on the previous model when building a new model for demonstrating and validating their technique in Aegis.

In the Aegis Sense-Control-Engage information pipeline, AWDRAT could be used to instrument some GUI (or other application) programs in the Control and Engage functions, e.g., operators that manage the process by negation as described earlier. If any deviations are detected, then those programs may have been compromised and they should be corrected. Similarly, Model-Based Executive could be used to instrument the Sensing systems, so their behavior can be observed and failures/deviations can be corrected automatically by software, complementing human operator oversight. This can be very useful when the number of incoming targets grows to a point of exceeding the human monitoring capability. Finally, Cortex tools may be used to instrument backend databases (e.g., the track file) for automated application tasks (e.g., for friend-or-foe recognition in a large coalition, mission planning with options, and workflow to obtain authorizations) to automate and mask the recovery from attacks or software failures of the database component.

BAA 06-35 mandates the response to attacks to be generated within 250 milliseconds of the attack. This is a significant challenge to the Cognitive projects, since that kind of response time is not necessarily an integral part of their original application environment (GUI, robot, and database). However, in principle the demonstration applications would not be prevented from being augmented with shortcuts that can achieve millisecond response for specific situations.

6.3.5. SRS Techniques in Insider Projects

[Functionality summarized in Section 3.5.5] In a protected environment such as the Aegis Sense-Control-Engage information pipeline, insider attacks are a major concern. Both PMOP and HDSM will monitor operator behavior, compare it against a model, learn from the history, and decide whether to allow or disallow anomalous behavior. The main challenge for the model-based approaches is their ability to adapt the approach and create a model for the prescribed Aegis scenario. The potential contributions of these projects to SRS Phase II seem to depend on whether they are able to leverage on the previous model when building a new model for demonstrating and validating their technique in Aegis.

PMOP has more explicit models of operational system, harm assessment, and intent assessment. These models distinguish (at an increasing level of sophistication) a malicious insider from operator errors. These models can be used to detect situations where a malicious insider operator may be trying to negate appropriate Aegis defensive actions through abnormal interference, so such non-authorized interference can be stopped. (This is a controversial topic since the issue of what an operator is or is not allowed to do has been debated for a long time, with valid arguments both ways.)

HDSM has a large sensor network and powerful high dimensional search engines that can learn and detect abnormal behaviors for a variety of applications and situations. Their demonstration includes human attackers penetrating machines to obtain sensitive military information or to launch distributed denial of service attacks. With sufficient training, these attacks can be detected at the initial stage (penetration) and stopped before that stage is successfully completed. The HDSM system can be demonstrated on a large distributed system such as Aegis Sense-Control-Engage information pipeline to detect any problems covered by its sensor network. However, the training period of HDSM probably needs to be done before deployment, since the millisecond timing constraints of Aegis would not allow real-time training.

6.3.6. Self-Regenerative Service for Aegis

The idea being pursued by the SRS team is the construction of a “self-regenerative service”, which would be a system of manageable size with API's, algorithm complexity, and quality of service requirements inspired by the needs of real systems. The outputs of this phase would be:

1. *evaluation of the prototype service against the motivating system-level goals of SRS regarding non-blink service and regeneration and automatic improvement over time,*
2. *assessment of the current SRS technical areas, and*
3. *pointers to additional technical areas, if necessary.*

If the prototype service is sufficiently similar to the class of combat systems in the most important aspects, hopefully the SRS team will at least gain evidence that SRS technologies can protect the real thing.

Prior work by DARPA and Dahlgren NSWC resulted in the HyperD combat system prototype, which is unclassified and believed to be unencumbered by proprietary restrictions. Some contractors of the SRS team worked on it in the Quorum program. Although the details need to be worked out, it looks likely that the SRS team can borrow code from HyperD for inclusion in the self-regenerative service. Ideally, the SRS team would work closely with the NSWC folks both to speed the team's own understanding of the combat system design/code and to keep the team aware of military needs.

Sections 6.3.1 through 6.3.5 have described some suggestions of how each SRS project may contribute useful techniques and tools for the Aegis scenario. In addition to self-regenerative functionality, we also have analyzed some of the techniques with respect to the severe Aegis timing constraints. Although a demonstration Self-Regenerative Service may not fit within the millisecond timing constraints of real-time Aegis Sense-Control-Engage information pipeline for a just-in-time regeneration process, the service can be used to pre-generate variants and pre-load these variants into the HyperD system. This was discussed briefly in Section 6.3.2 and is repeated below for the Self-Regenerative Service.

The millisecond-range real-time requirements can be a significant challenge to the Self-Regenerative Service and SRS tools, since the regeneration process needs to be done somehow within the severe time constraints. The current setup of program variant replacement in many projects (including Diversity, Redundancy, Cognitive, and Insider) seem to require loading the new variant from a file system and re-initialization of the new variant. As an optimization to bypass the loading phase latency, the new variants will need to be pre-loaded into main memory, and activated as needed (a warm standby). As a further optimization, these new variants may be executed concurrently (a hot standby) and results voted, if full replicated and concurrent processing is required. This would be the case if the re-processing would cause the system to miss its original deadlines. Concurrent processing may also be desirable simply to speed up the Sense-Control-Engage pipeline, since an incoming target must be dealt with as soon as possible.

6.4. DDX Application Scenario

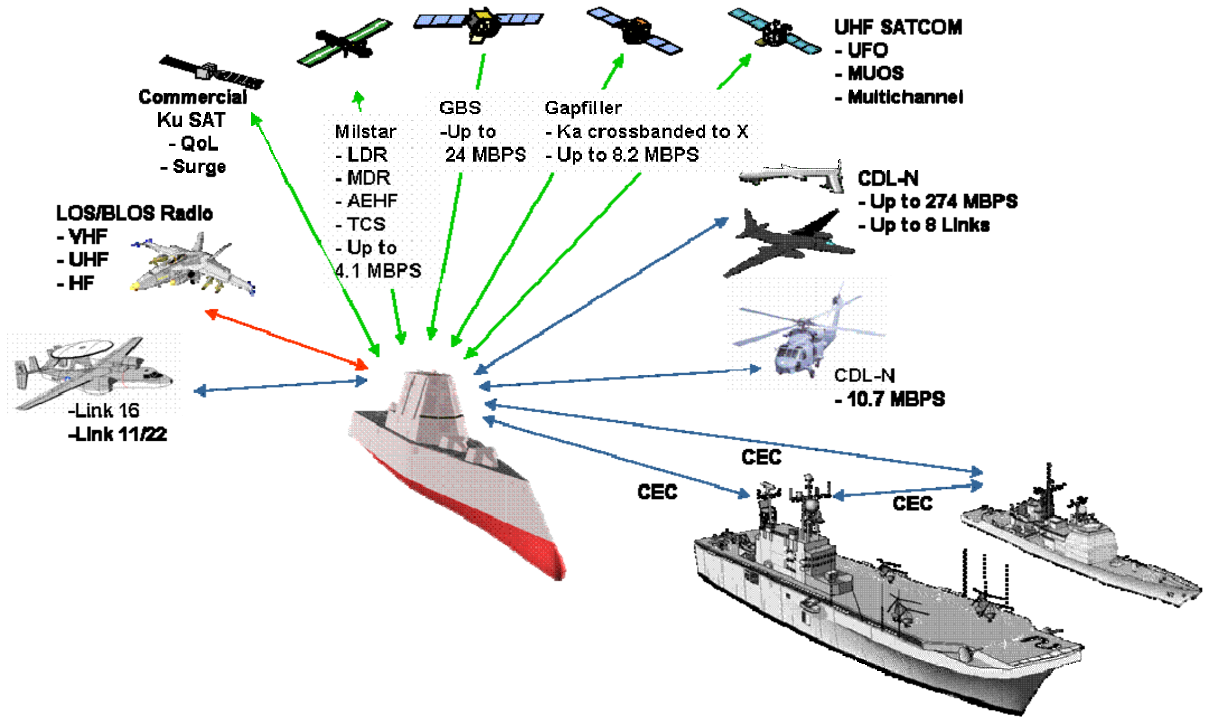


Figure 13 DD(X) Communications Scenario

Another scenario that we have started studying is Navy's DD(X) Total Ship Computing Environment. A DD(X) ship contains 3 data centers and highly redundant networking connections. It is a highly survivable environment by design. It would seem that SRS technologies would be able to help improve the performance and survivability of DD(X) computer systems. The concrete scenario is still being developed at this writing (February 21, 2006).

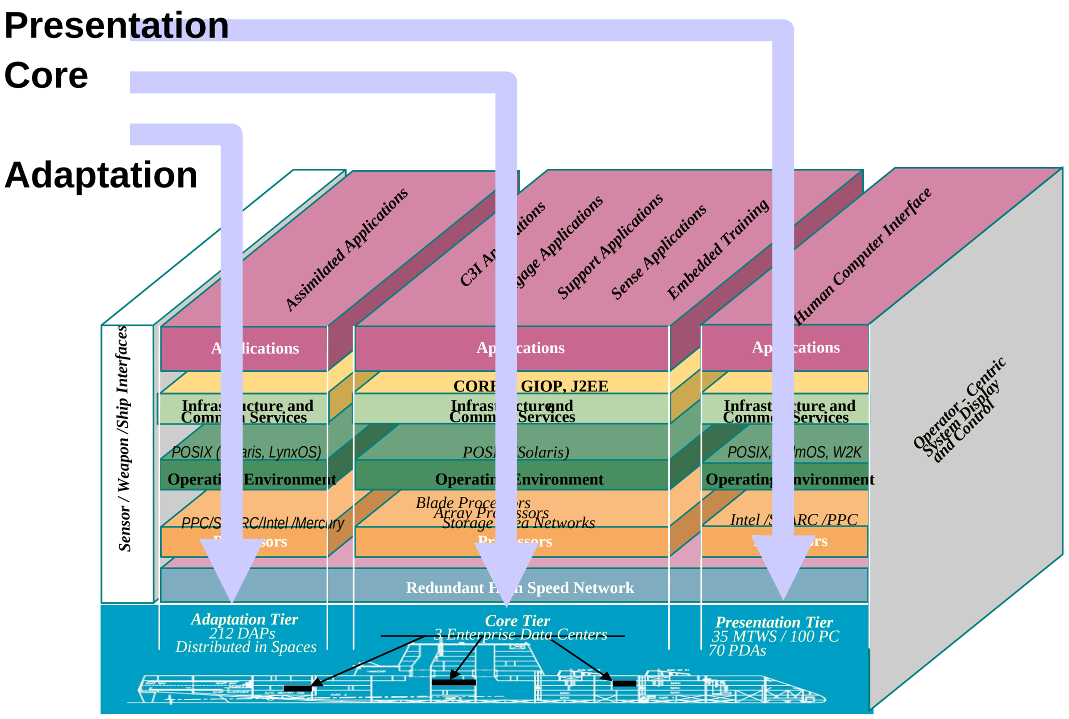


Figure 14 DD(X) Total Ship Computing Environment

The DD(X) Total Ship Computing Environment (TSCE), shown in Figure 14, contains many redundant components. The Self-Regenerative Service may be able to increase the degree of redundancy and availability in TSCE without increasing the complexity of the original software system.

7. Acknowledgements

Section 2 and Sections 3.1 through 3.4 are adapted from the Part 1 report of this architecture study. Section 3.5 contains some terms (in bold face) quoted from the document “Integrated SRS Defense”, developed by GlobalInfoTek and Bob Balzer. Section 6.1 scenario on Surface Combat System was developed and distributed by Lee Badger. The figures on DD(X) in Section 6.4 are from a presentation provided by Brett Chapell.

APPENDIX A:
Part 1 of RSRS Architecture Study (Progress Report)

Purpose of Report

This report is the project deliverable summarizing the total effort expended by the Georgia Institute of Technology in support of Griffiss Air Force Base and DARPA on Grant FA8750-05-1-0253, supported by the SRS program.

The main goal of RSRS architecture is a conceptually simple, but functionally rich description of the four topic areas of SRS program as well as the interactions among them. Specifically, we use the concepts of reflection and feedback control in the SRS components and between components. The feedback control is part of monitor-learning-actuator (MLA) loop. By allowing recursive and mutual invocations of the SRS components and the presence of MLA loops at different component levels, we will be able to create a rich set of capabilities that demonstrate the power of an SRS application or system. In the proposed study, we will develop the RSRS architecture in detail, describing the internal structure of each component and the mutual invocations among the components.

Members of the Project: Prof. Calton Pu and Prof. Douglas Blough.

Reflective Self-Regenerative Systems (RSRS) Architecture Study

Part 1: Architectural Analysis and Evaluation of SRS Projects

Calton Pu {calton@cc.gatech.edu} and *Douglas Blough* {doug.blough@ece.gatech.edu}
Georgia Institute of Technology

A. Summary

Architecture. The RSRS Architecture is a conceptually simple, yet functionally rich description of the four topic areas of SRS program, as well as the interactions among them. RSRS uses the concepts of reflection and feedback control between SRS components and within each component to describe their internal structure and external interactions. The main feature of RSRS is the monitor-learning-actuator (MLA) loop, based on the concepts of reflection and feedback control. The MLA is present in each system layer and component that supports self-regeneration. By monitoring component and system behavior, systems and applications are able to detect attacks and recover through the regeneration of data and programs using diversity to counter attacks targeting the physical representation of application (or system) programs. The MLA abstraction also suggests a standard interface among the technology areas as well as supporting customized extensions for each project. In addition to the main MLA abstraction, RSRS also includes component tools for specific areas such as Diversity and Redundancy.

Interfaces. The RSRS description of component interactions can be captured by a set of standard interfaces. Such a standard interface defines the common functionality among the projects of each area. Furthermore, extensions to these interfaces can be designed to support unique features provided by individual projects. The RSRS interface design is based on events, which carry both significant state information (the monitoring part of MLA) and control commands (the actuator part of MLA). The state notification events have a high degree of composability between event types. The control events enable various feedback control mechanisms to be adopted, including distributed and centralized control. These interfaces will support a manageable integration of SRS projects and provide the SRS facilities to other applications and demonstration projects that need self-regenerative capabilities. A high-level specification of the general RSRS interface is given in Part 2 of this study. Development of detailed interfaces is premature at this stage, because most of the software products developed in Phase 1 are not designed as components with well-defined external interfaces.

Evaluation. In this report (Part 1), the RSRS architecture has been successfully applied to model the self-regenerative aspects of current SRS projects. These models can be used to compare the projects of the same area at a qualitative level. A quantitative evaluation of SRS projects is pending due to the in-progress status of the projects and their ongoing evaluation by the Red Teams. In the next report (Part 2), we will look forward and attempt to use the lessons in this architectural study to fit the outcome of the SRS projects together.

B. RSRS Architecture

B.1 RSRS Main Concepts

The main concepts used in RSRS (see Figure 1) are: (1) feedback control and reflection in the MLA loop for self-regeneration, and (2) recursive and mutual use of component tools such as Diversity and Redundancy. In Figure 1, we have liberally added recursive use of tools among the four key technology areas of SRS: (1) Biologically-Inspired Diversity, (2) Cognitive Immunity and Regeneration, (3) Granular, Scalable Redundancy, and (4) Reasoning About Insider Threats.[2] This level of interaction and integration may not be achieved at the end of Phase 1; however, an effective architecture for the SRS program must show the potential interfaces for integration among the tools and techniques being developed by these technology areas.

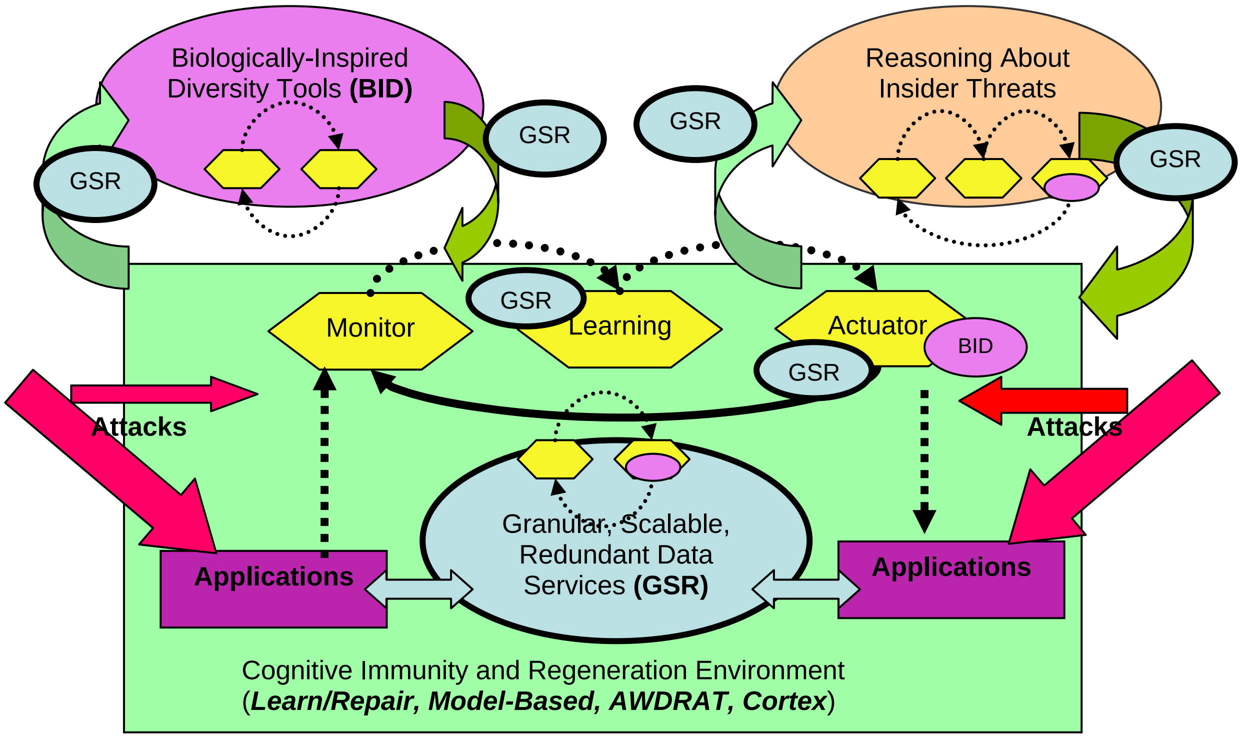

Biologically-Inspired Diversity Tools (BID)
Reasoning About Insider Threats
GSR
GSR
GSR
GSR
GSR
GSR
Monitor
Learning
Actuator
BID
Attacks
Attacks
Granular, Scalable, Redundant Data Services (GSR)
Applications
Applications
Cognitive Immunity and Regeneration Environment (Learn/Repair, Model-Based, AWDRAT, Cortex)

Figure 15 RSRS Functional Architecture

MLA Loop. The foundation of RSRS is the monitor-learning-actuator (MLA) loop, based on the concepts of feedback control in engineering systems and reflection in programming languages. The MLA loop consists of three components; (1) a monitor that reflectively observes system behavior (e.g., in applications running inside a Cognitive Immunity and Regeneration environment) or watches for anomalies in other parts of the system (e.g., insider threat monitors), (2) an actuator that takes self-regenerative action to recover from observed anomalies, and (3) an optional learning component that records events observed by the monitor and actions taken by the actuator, and manages knowledge for better decision making in self-regeneration actions.

[2] For brevity and where there is no ambiguity, we will refer to area (1) as “Diversity”, area (2) as “Cognitive”, area (3) as “Redundancy”, and area (4) as “Insider”.

Role of MLA. The MLA loop can be found in all four technology areas of the SRS program. The MLA loop is represented in Figure 1 as yellow hexagons linked by connecting arrows. The loop is most prominent in the Cognitive[1] area, where four projects are building environments that support the monitoring, learning, and self-regeneration of application and system components to make them resilient against attacks. The MLA loop is also a major structural component of the two projects in the Insider[1] area, since such systems must observe the system behavior to detect and then counter insider attacks. The loop is more implicit in the two "service" technology areas. For example, the three projects in the Redundancy[1] area provide redundant data delivery services to improve the availability and reliability of the entire system. It is natural that these data services utilize MLA tools reflectively to improve their own availability and reliability. Similarly, the two projects in the Diversity[1] area, which create program and data generation tools, can use MLA to avoid monoculture vulnerabilities in their own tools.

The MLA loop primarily captures the self-regenerative aspect of SRS projects. In the Diversity and Redundancy areas, the special capabilities provided by the projects are not necessarily self-regenerative. When discussing the inherent (non-self-regenerative) functionalities of components in these areas, we refer to them as "tools".

Diversity Tools. In the Diversity area, the projects will provide tools to create variants of binary representations of programs. These variants should have sufficient differences among them so attacks that depend on specific bit-layout (e.g., typical buffer overflow attacks) would not work on all variants. The variants are used by self-regenerative programs (e.g., in the Cognitive area), but the tools that generate the variants may be passive. The degree of difference among the variants (and the associated resistance to attacks) is outside the scope of the MLA loop model.

Redundancy Tools. Similar to the Diversity Tools, the Redundancy area provides tools for data replication, object replication, and reliable communication, which have the capability to resist certain types of attacks through redundancy. The performance and scalability of the specific redundancy mechanisms, while important to the SRS program, is outside the scope of the MLA loop model.

The rest of the report is organized as follows. Sections 2.2 through 2.5 apply the MLA loop to analyze the self-regeneration aspects of the four SRS areas. Section C contains an architectural evaluation of the current projects in each area. Section D concludes the report with a self-evaluation of the RSRS architecture.

B.2 RSRS Analysis of Cognitive Projects

Cognitive MLA. In the Cognitive Area, the self-regenerative healing process can be described using the MLA loop. The three phases of self-regeneration (monitoring, learning, and regeneration) are outlined in this section and illustrated in Figure 2.

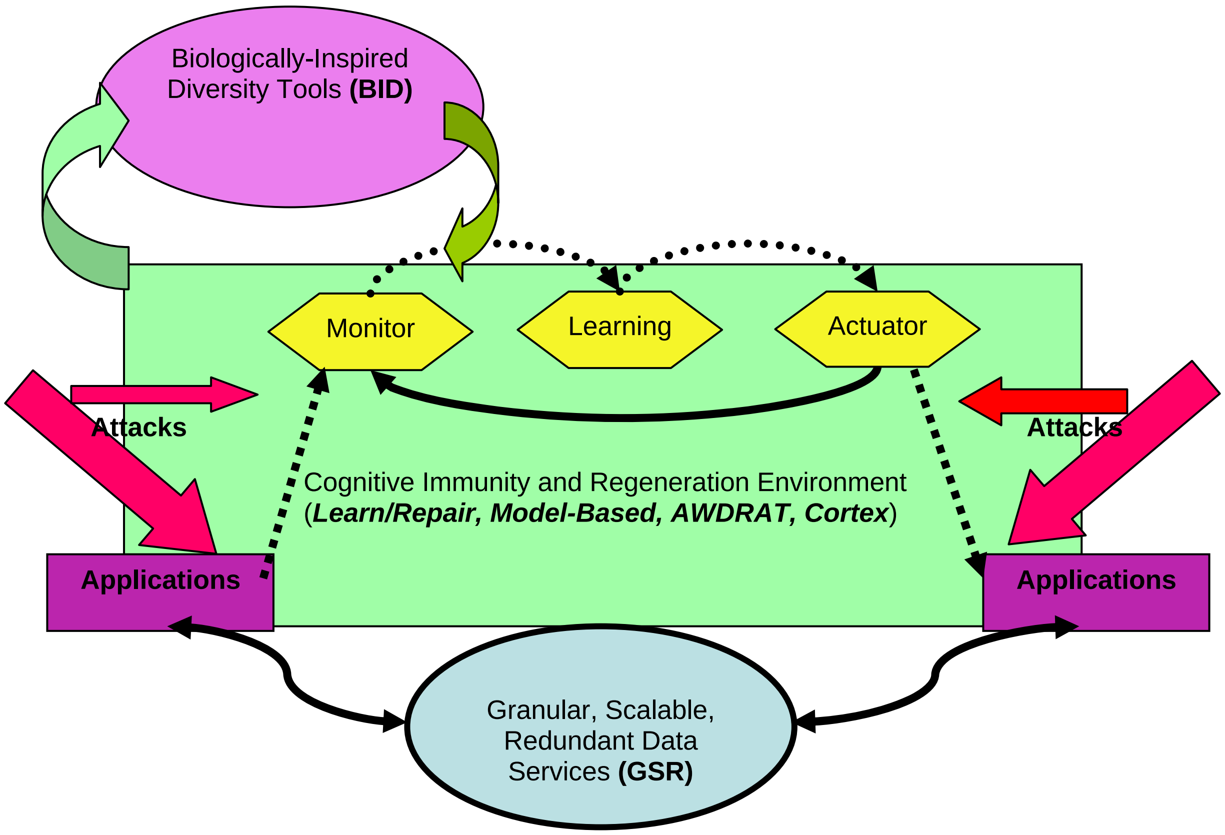

Biologically-Inspired Diversity Tools **(BID)**
Monitor
Learning
Actuator
Attacks
Attacks
Cognitive Immunity and Regeneration Environment (***Learn/Repair, Model-Based, AWDRAT, Cortex***)
Applications
Applications
Granular, Scalable, Redundant Data Services **(GSR)**

Figure 16 RSRS Architecture for the Cognitive Area

Monitoring. The first part of the Cognitive MLA is a monitoring service, which will monitor events at several time granularities and at different levels of system services. This approach comes from the observation that system failures and malicious attacks may occur through events at all time scales and system levels. Consequently, it is necessary for an SRS system to monitor the vital signs and anomalous events at several appropriate time scales and levels of abstraction. At each level of time scale and abstraction (e.g., hardware, kernel, middleware, and application process), the monitoring service will observe meaningful system states, compare them to acceptable states (using potentially different models and techniques), and generate events if significant state changes are detected.

Learning-Based Diagnosis. The second part of the Cognitive MLA is a learning-based diagnosis service, which may vary according to the time scale of events and their level of abstraction. For example, diagnosis at the hardware and kernel levels may be based on control systems. In contrast, diagnosis at the middleware and application levels may rely on model-based reasoning or data mining techniques. We use an abstract learning process to model the various learning-based diagnosis services. Abstractly, the events observed by the monitoring service are stored in an Events Database (ED), and the diagnosis process is a comparison of patterns in the ED with patterns previous learned and stored in a Known Events Database (KED). The KED stores both significant event patterns (problems) and an appropriate response and defense for those problems. In model-based systems, the KED captures the knowledge represented by the models. The learning part of MLA is represented by addition of knowledge into KED.

Regenerative Actuation. The third part of the Cognitive MLA is a regenerative actuator, which finds and carries out the recovery actions specified by the KED. For example, software updates may be available for a known virus using buffer overflow. In this case, KED will contain the recipe to apply the software update, or send an updated copy of system software to the affected node. If the problem is unknown (e.g., a new DoS or virus attack), then the Cognitive MLA enters a sub-loop to build and find an effective regenerative action using two high level services: a program diversity service provided by the Diversity Area and a data redundancy service provided by the Redundancy Area. The sub-loop will identify the extent of damage, create new system images to repair the damaged components, test them against the attack, find the variants that are effective against the attacks, store them in the KED, and distribute these variants in regenerative actions.

The sub-loop to search for remedies builds on the other SRS areas. For example, the new system image variants are created by program diversity tools from the Diversity Area. These variants may have been preventively generated beforehand, or dynamically generated at run-time. Similarly, trusted data and communications are provided by data redundancy services from the Redundancy Area. Each variant is then tested by creating an environment with the new variant and exposing it to the environmental conditions during the attack. If a new variant is shown to be resistant to the attack, it is entered into the KED and used to recover from the attack. The entire sub-loop may be online or offline, depending on the knowledge contained in the KED and the policies for recovery.

Cognitive Projects. Sophisticated software tools are being developed by the Cognitive area projects (***Learn/Repair, Model-Based, AWDRAT, Cortex***) that can be described by MLA loops. All of these projects have significant R&D efforts in monitoring, learning, and repairing of applications that run in their own environments. Although these efforts are currently isolated in their own projects, their components may be made available and interoperable through a standard interface based on the MLA abstraction. These tools can then be adopted by and integrated with other technology areas. Three of the four Cognitive area projects (*Model-Based, AWDRAT, Cortex*) employ model-based approaches where self regeneration is triggered by detection of deviations from a predictive model of the system. We note that the model-based approach is a specialization of the MLA approach, where in the monitoring component, system behavior is compared against model outputs, in the learning component, the system model is dynamically updated based on actual system outputs, and in the actuator component, adaptation of the system is performed based on model deviations.

B.3 RSRS Analysis of Diversity Projects

The Diversity area (*Genesis* and *Dawson* projects) develops code and data diversification tools that will be used by other areas and applications when attacks are detected. In Figure 1, the Diversity tools are represented as purple ovals, typically used by the actuator component of an MLA loop.

Figure 3 shows the RSRS architecture view of Diversity projects.

The main goal of Diversity projects is to develop Diversity Component Tools (algorithms and software) to achieve design diversity and data diversity goals. To measure the effectiveness of diversity algorithms, the evaluation steps can be modeled by an MLA-style loop.

- Monitoring: After the variants are created, their resistance to attacks is evaluated.
- Learning-Based Diagnosis: The winning variants are stored in a KED, while the losing variants are marked as such or discarded.
- Regenerative Actuation: The winning variants are then used to increase system robustness by replacing vulnerable components, possibly by a Cognitive component or system.

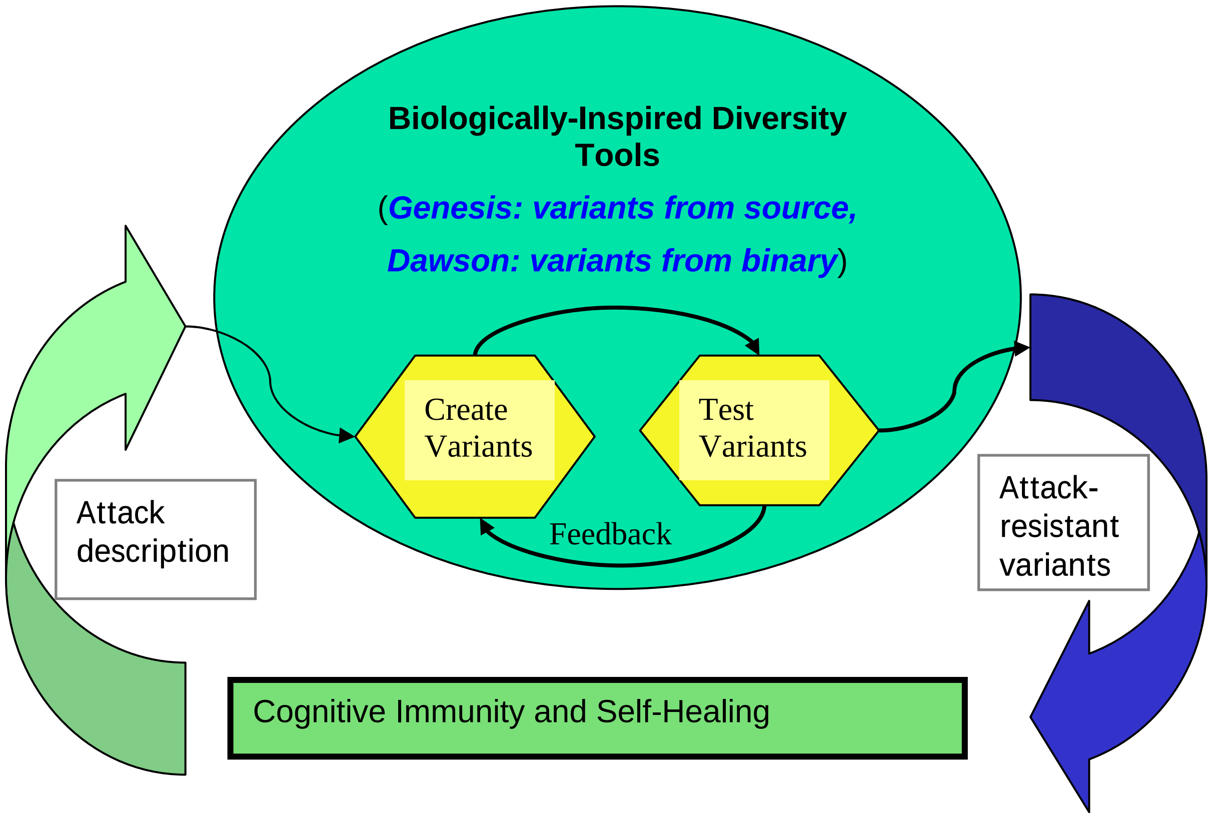

Biologically-Inspired Diversity Tools
(Genesis: variants from source,
Dawson: variants from binary)
Create Variants
Test Variants
Feedback
Attack description
Attack-resistant variants
Cognitive Immunity and Self-Healing

Figure 17 RSR Architecture of Diversity Area

B.4 RSRS Analysis of Redundancy Projects

The Redundancy area projects develop highly-available services for communication, data storage, and computation for use by other areas and applications. For example, applications that run in a Cognitive Regeneration environment may have their communications and/or data flowing through channels created by the ***SAIIA***, ***IITSR***, or ***QuickSilver*** projects.

Figure 4 shows the RSRS architecture view of Redundancy projects.

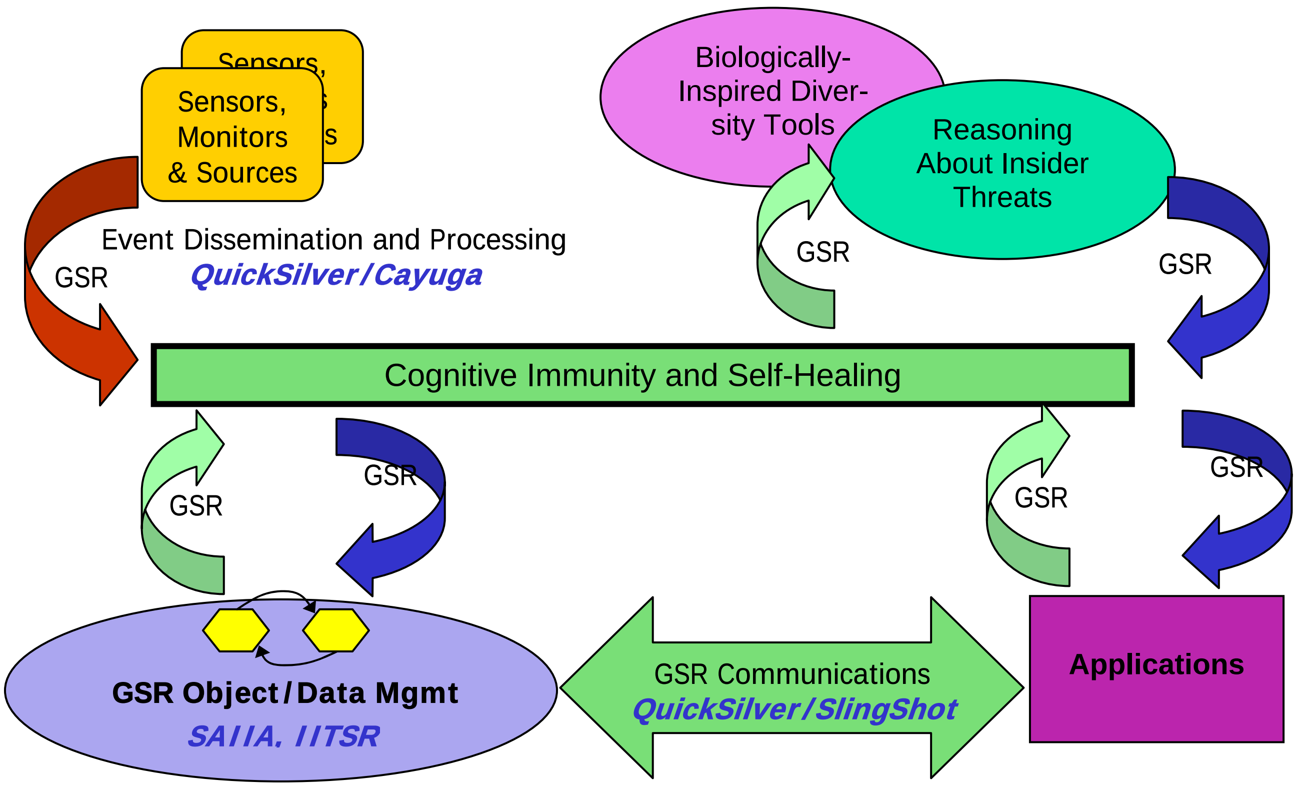

Sensors, Monitors & Sources
Sensors, Monitors & Sources
Biologically-Inspired Diversity Tools
Reasoning About Insider Threats
Event Dissemination and Processing
QuickSilver/Cayuga
GSR
GSR
GSR
Cognitive Immunity and Self-Healing
GSR
GSR
GSR
GSR
GSR Object/Data Mgmt
SAIIA, IITSR
GSR Communications
QuickSilver/SlingShot
Applications

Figure 18 RSR Architecture of Redundancy Area

B.5 RSRS Analysis of Insider Projects

The Insider technology area (*PMOP* and *MIT-HDSM* projects) consists of meta-level software tools that monitor system-level events and compare these events to a correct behavioral model or acceptable norm. Their MLA loops contain significant monitoring components (many system and application events may be relevant), learning components (dynamic adaptation may be required when under insider threats, e.g., temporary adoption of more conservative security rules), and actuator components (e.g., generating and switching to a different set of program modules that implement a more restrictive set of rules to counteract the suspected threat). The Insider area projects are represented as peach-colored ovals in Figure 1.

Figure 5 shows the RSRS architecture view of Insider projects.

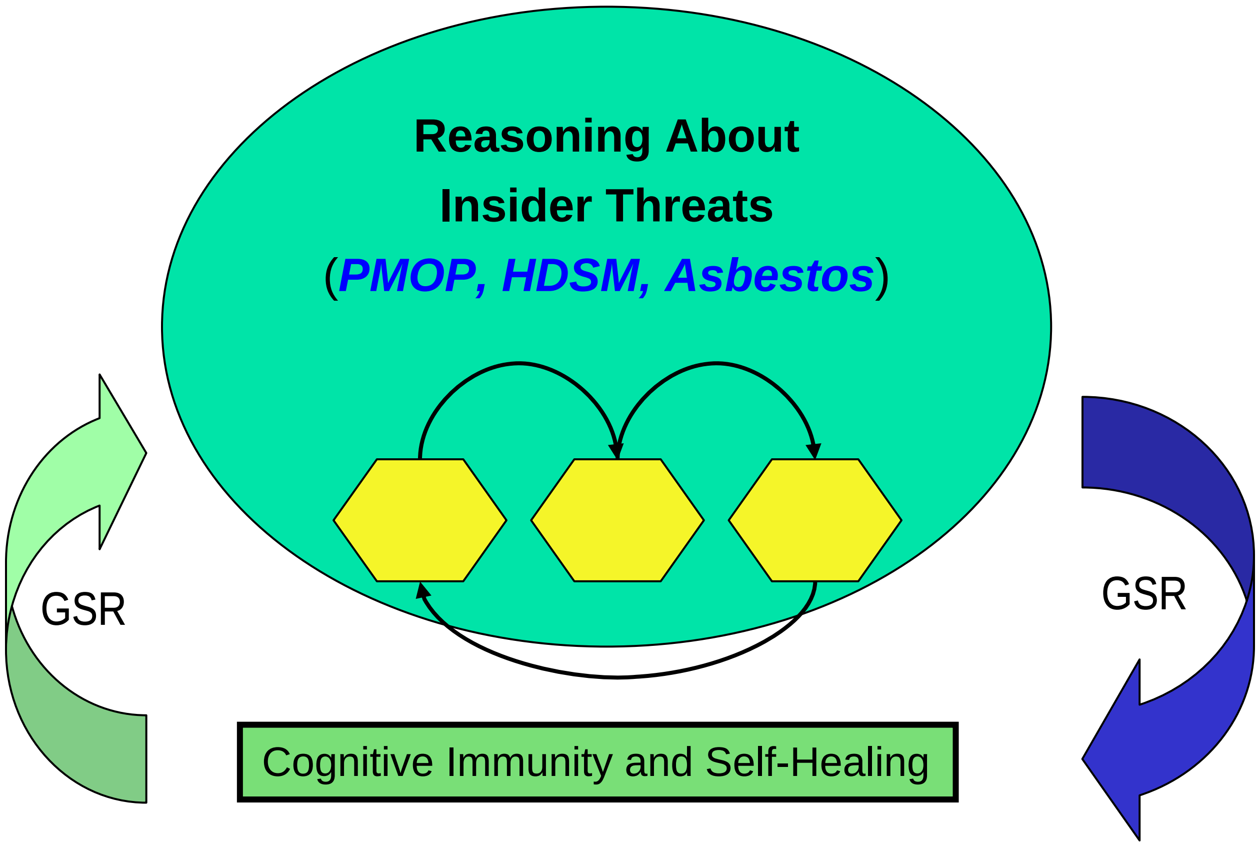


Figure 19 RSRS Architecture for the Insider Area

C. Architectural Evaluation of SRS Projects

In this section, we apply the RSRS architecture to the projects in each area, by translating each project's architecture into the general RSRS structure shown from Figure 1 through Figure 5.

- The Cognitive projects in Section C.1 correspond to the projects represented by Figure 2 RSRS Architecture for the Cognitive Area.
- The Diversity projects in Section C.2 correspond to the projects represented by Figure 3 RSRS Architecture of Diversity Area.
- The Redundancy projects in Section C.3 correspond to the projects represented by Figure 4 RSRS Architecture of Redundancy Area.
- The Insider projects in Section C.4 correspond to the projects represented by Figure 5 RSRS Architecture for the Insider Area.

C.1 RSRS Architectural Evaluation of Cognitive Projects

C.1.1 Learning and Repair Techniques for Self-Healing Systems

Learn/Repair (MIT: Michael Ernst and Martin Rinard)

Daikon is a software tool that verifies and preserves invariants of program data structures. Programmers specify the invariants that data structures should maintain during execution. Daikon creates the code that monitors the invariants during execution and repair algorithms that attempt to restore the invariants.

Figure 20Figure 20 shows the RSRS architecture evaluation of the Learn-and-Repair project, by describing the Daikon facilities using MLA concepts. Examples of Daikon techniques modeled using MLA concepts include:

- Monitoring: In an ahead-of-time step, the target program is executed under a special run-time system that observes the values of program variables.
- Learning-Based Diagnosis: Machine learning is performed over the observed variable values, producing a model (a set of data structure constraints). The data structure repair compiler automatically compiles a repair strategy for these properties into the target program.
- Regenerative Actuation: If any constraint is violated at run-time, the repair algorithm performs goal-directed planning to create a repair plan, consisting of a sequence of individual data repair actions. Each data repair action modifies a data structure to re-establish the constraints.

Concrete deliverables from the Learn/Repair project are:

- the Daikon tool for dynamic invariant (model) inference
- a data structure repair compiler
- a data structure repair language (supported by the tools)

Summary of Learn/Repair Evaluation. The Learn/Repair project has a clear self-regenerative nature in its MLA feature, as shown in Figure 20. The scope of the Daikon tool is within a program, concerning the regeneration of decayed data structures. This technique can be applied by application programs to become more robust against internal data structure corruption. This is the only project working on self-regenerative techniques applied inside a program.

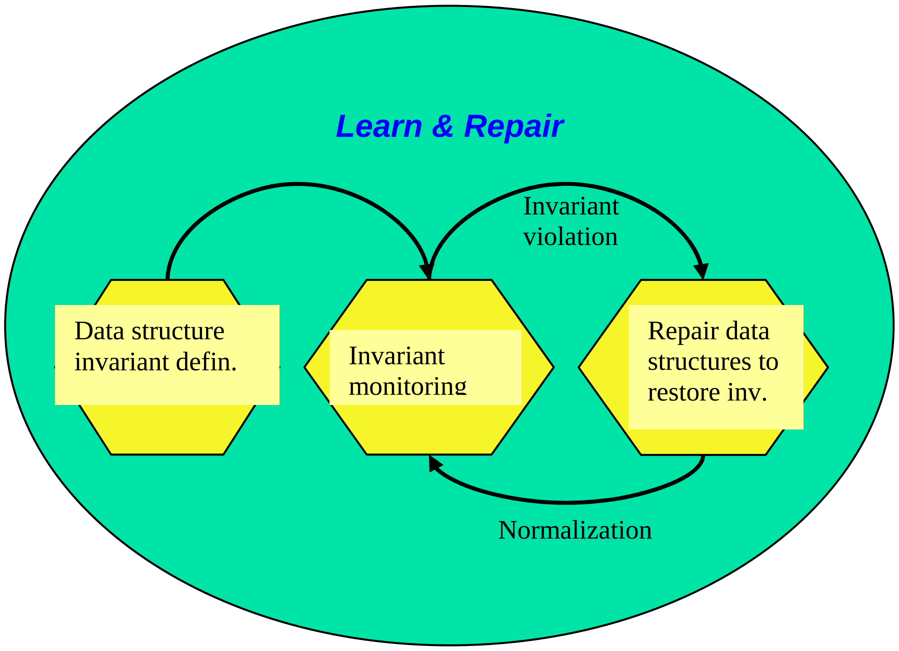

Learn & Repair
Invariant violation
Data structure invariant defin.
Invariant monitoring
Repair data structures to restore inv.
Normalization

Figure 20 Main MLA Features of Learn-and-Repair

C.1.2 Pervasive Self-Regeneration through Concurrent Model-Based Execution

Model-Based Execution (MIT: Brian Williams and Gregory Sullivan)

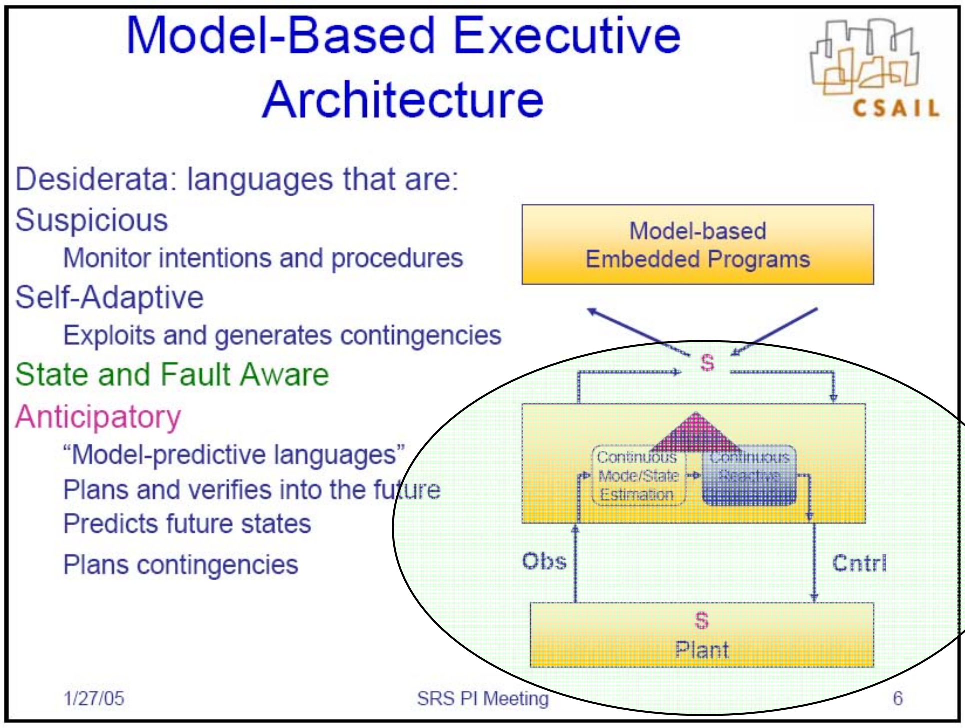
Summary: Model-based execution can be modeled as an MLA loop, as follows.

- Monitoring: Pervasive system robustness by composing concurrent fault aware processes. Self-deprecating methods through prognostic mode estimation.
- Learning-Based Diagnosis: Fault-adaptive processes through model-based program execution. (The model-based executive can construct *novel recovery actions* in the face of *novel faults.*) Safe fault adaptation through method dispatch as continuous planning. Incorporation of fault adaptation incrementally.
- Regenerative Actuation: Self-regenerating methods through redundant method dispatch. Self-optimizing methods through decision-theoretic dispatch.

Figure 21 illustrates the architecture of the project, taken from the presentation in the January 2005 SRS PI meeting. It shows a MLA-style feedback loop in their architecture, highlighted with the light green oval.

Figure 22 shows the RSRS analysis of the main MLA functions of their project. The experimental part of the project (the robotics application) will demonstrate the learning and regenerative capabilities of the approach in the application domain. . Concrete deliverables from the project are the RMPL language and the MIT_MERS rover testbed.

Summary of Model-Based Execution Evaluation. The Model-Based Execution contains a monitor that detects faults during execution by comparing the execution results against the predictions of a model. Their Learning-Based Diagnosis module can define novel recovery actions for novel faults. The system method dispatch module can cause methods to be self-regenerated through redundancy and/or self-optimized through decision-theoretic approaches. This method may be applied to any application for which appropriate models can be created.



Model-Based Executive Architecture

Desiderata: languages that are:

Suspicious
- Monitor intentions and procedures

Self-Adaptive
- Exploits and generates contingencies

State and Fault Aware

Anticipatory
- "Model-predictive languages"
- Plans and verifies into the future
- Predicts future states
- Plans contingencies

1/27/05 SRS PI Meeting 6

CSAIL
Model-based Embedded Programs
S
Continuous Mode/State Estimation
Continuous Reactive
Obs
Cntrl
S
Plant

Figure 21 Structure of Model-Based Executive Architecture

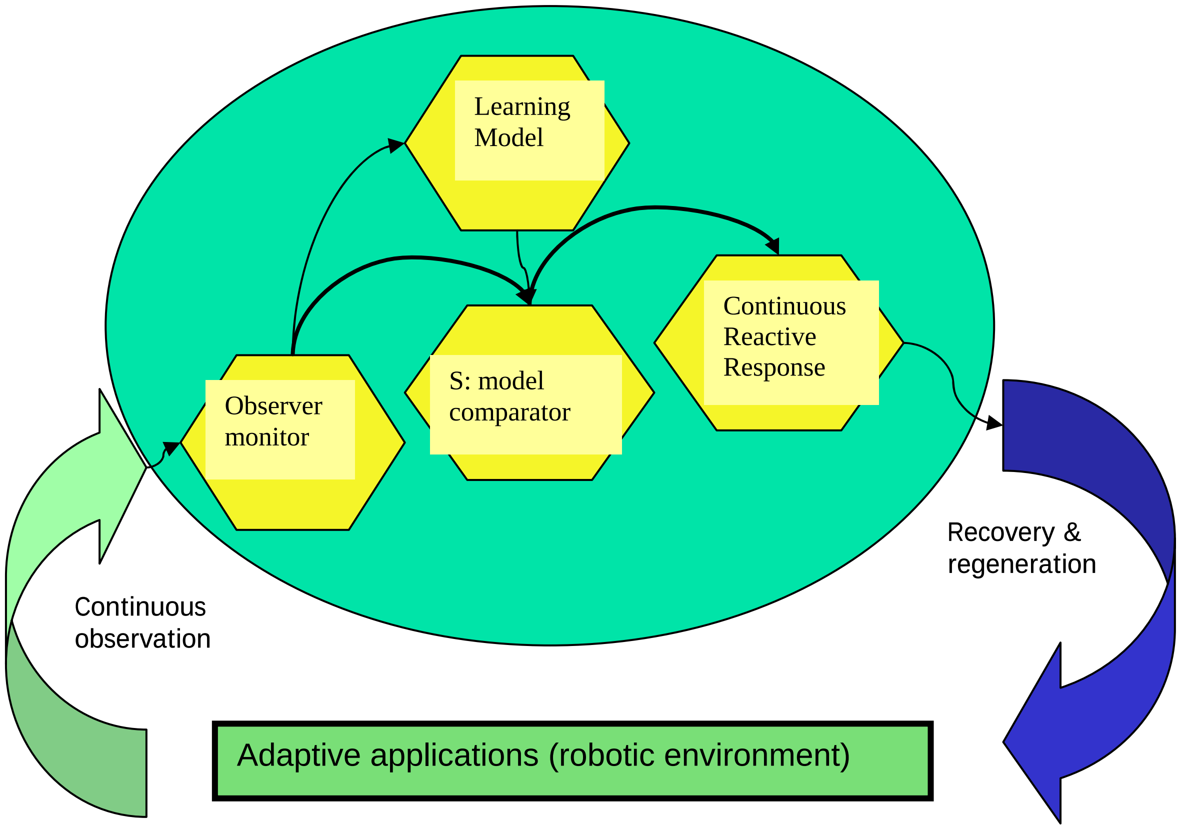


Figure 22 Main MLA Functions of Model-Based Executive

C.1.3 Architectural Differencing, Wrappers, Diagnosis, Recovery, Adaptivity, and Trust Modeling

AWDRAT (MIT: Howie Shrobe; Teknowledge: Bob Balzer)

Figure 23 illustrates the structure of the AWDRAT toolkit, taken from one of the slides of the AWDRAT presentation at 07/12/05 PI Meeting. The AWDRAT approach generates wrappers that monitor application software execution. The wrappers and other sensors observe the important execution parameters, comparing them to a canonical system model through a process called Architectural Differencing. If sufficient deviation is found between the specified constraints of the architectural model and execution parameters, then an attack is recognized and recovery actions initiated. The recovery actions include restoration of databases, code segments, and other recoverable data. Appropriate methods (possibly created by other projects such as Genesis and Dawson) are combined with consistent data in a new execution.

Experiments. The AWDRAT toolkit will be demonstrated and evaluated in OASIS DEMVAL system, specifically the MAF/CAF component.

From the RSRS architecture point of view, the AWDRAT decision cycle shown in Figure 23 can be modeled as an MLA loop as follows:

- Monitoring: *Architectural Differencing* - Accompanying each method is an architectural model of the computation performed by the method. AWDRAT interprets this in parallel with the executing code, using wrappers to extract data from the method's execution and noting when the executing code violates a constraint of the architectural model.
- Learning-Based Diagnosis: If any constraint imposed by the architectural model is violated, model-based diagnosis is invoked to assess what part of the computation may have failed and the trust model is updated with the information produced by the diagnosis, leading to new assessments of the trustability of the computational resources.
- Regenerative Actuation: Recoverable data (e.g. databases, code segments, password files) are restored in order to establish a consistent point from which to resume the computation. AWDRAT consults its method library, finding all applicable methods relevant to the service request. Each combination of method and supporting resources is evaluated, taking into account the cost of the resources and the value of the service quality delivered.

Figure 24 shows the RSRS architecture evaluation of AWDRAT, by transforming the structure shown in Figure 23 into the RSRS architecture concepts as explained above. We can see the main input sources (data produced by the sensors and the wrapper monitor on the execution side and the system models on the specification side) of the Architectural Differencer. The Architectural Differencer determines whether an attack situation should be evaluated by the Trust model diagnosis module, which is capable of learning from new cases and feeding the augmented model into subsequent evaluations.

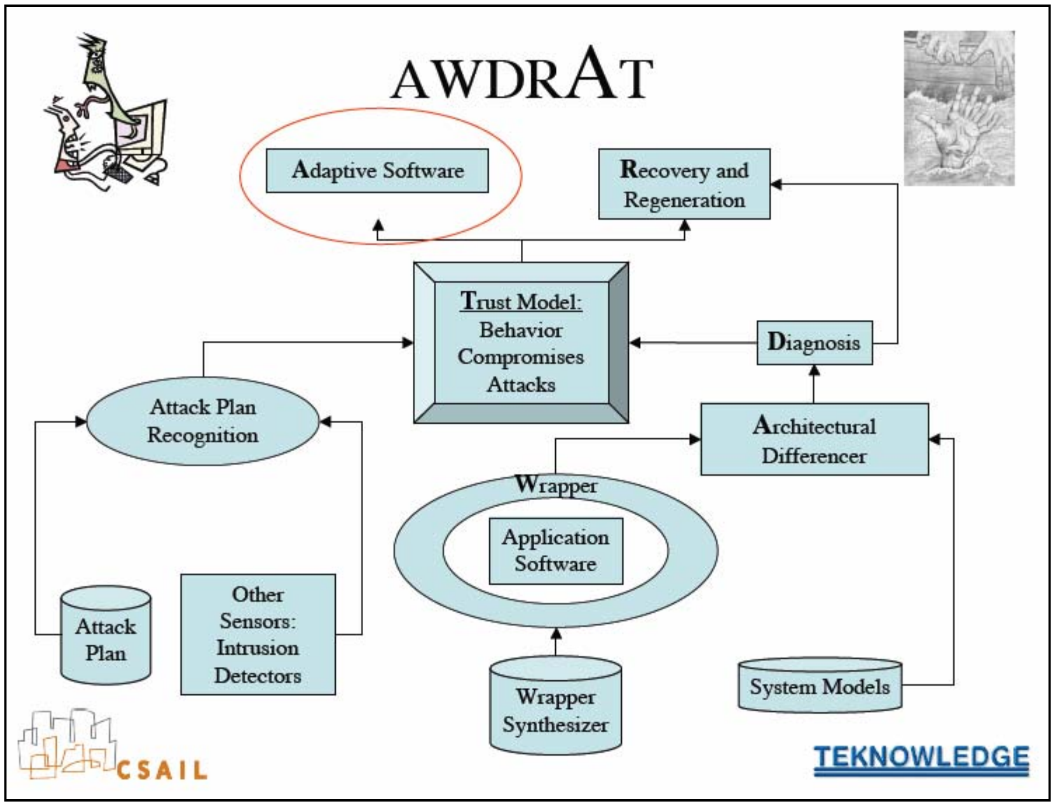


Figure 23 Structure of AWDRAT Toolkit

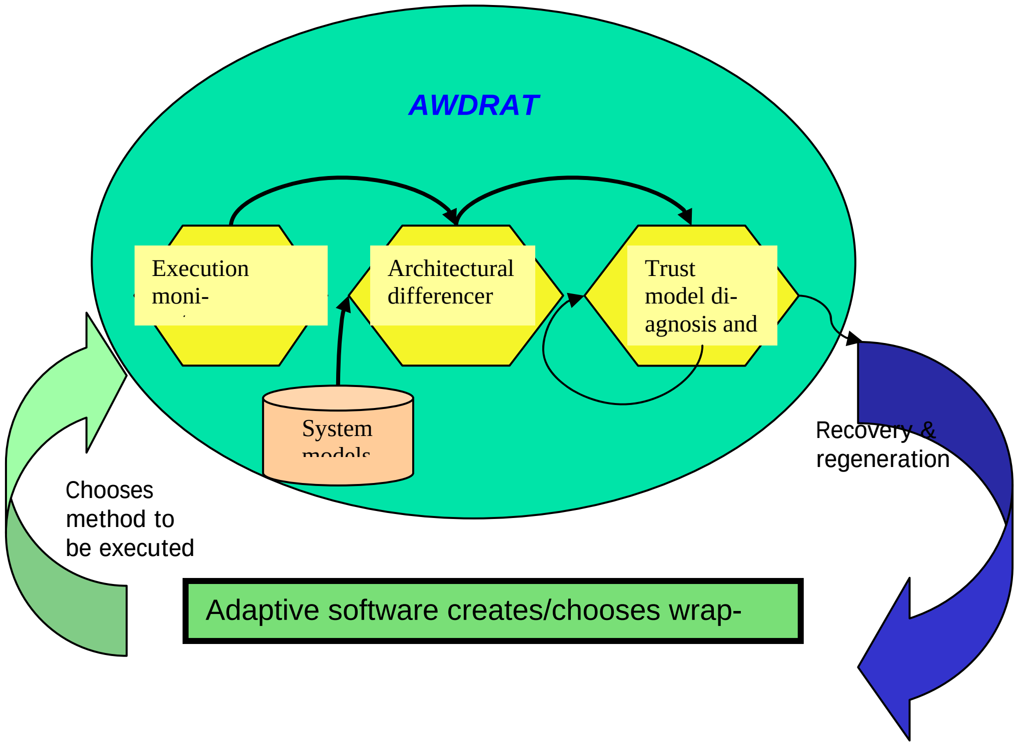

AWDRAT
Execution moni-
Architectural differencer
Trust model di- agnosis and
System models
Recovery & regeneration
Chooses method to be executed
Adaptive software creates/chooses wrap-

Figure 24 Main MLA Features of AWDRAT

There are several levels of MLA loops in the AWDRAT project. At the module level, the learning MLA is represented by an arrow looping back to the Trust Model and Diagnosis module. At the system level, the normal execution flows through the system without further actions. If attacks are detected by Architectural Differencer and confirmed by the Trust Model, the recovery and regeneration actions are initiated. Adaptive Software modules restore data consistency and choose appropriate wrappers and methods for the new execution, represented by the black arrows outside the inner MLA loop. This “outer” MLA loop is currently part of the AWDRAT project, although in principle it could be done by software tools created by another of the Cognitive projects (Section C.1) that work at the system level.

Summary of AWDRAT Evaluation. The AWDRAT system builds an architectural model of each method executed. During the method execution, the model is evaluated concurrently and a monitor called Architectural Differencer compares the model against the method execution results. If the Differencer finds discrepancies, the learning-based diagnosis module updates the trust model for future reference. If damage is detected after differences are found, the regenerative action restores necessary data to a consistent state. Their tools can be applied to any application for which appropriate models can be defined for each potentially vulnerable method.

C.1.4 Cortex

Cortex (Honeywell: David Musliner)

Figure 25 illustrates the structure of Cortex Demo Architecture, taken from one of the slides of the Cortex presentation at 07/12/05 PI Meeting. The Cortex project is building a MySQL proxy that can filter incoming SQL database queries by comparing them with known exploits from snort's rules and, new rules from Cortex's learning module. One of the learning techniques employed by Cortex is the use of “Taster Databases” that run the unknown queries as a test. If the queries turn out to be malicious, then they have compromised only a Taster Database. A Replicator module maintains the consistency between multiple Taster Databases and the Master Database.

Experiments. Cortex is using MySQL as the demonstration application. They have built a prototype and evaluated the system response to known attacks (buffer overflow and illegal parameter). The Cortex system is able to defend itself against these attacks.

From the RSRS architecture point of view, the Cortex architecture can be modeled as an MLA loop as follows:

- Monitoring: *Scalable Coherent State Estimation* - Cortex will use highly scalable qualitative probabilistic algorithms to combine the noisy, uncertain outputs from numerous system sensors into an accurate and coherent estimate of system state.
- Learning-Based Diagnosis: *On-line Learning* - Cortex will begin operations with models of the computing system it controls, its mission, and the faults and attacks that may disturb it. Over time, Cortex will use statistical and structural learning algorithms to continually refine those models, improving the accuracy of its self-awareness and mission-awareness.
- Regenerative Actuation: *Mission-Optimized Planning and Response* - Cortex will use cognitively-inspired mission-aware planning algorithms to derive proactive response plans that optimize the system's mission performance during disruptions and attacks.

Figure 26 shows the RSRS architecture evaluation of Cortex, by translating the structure shown in Figure 25 into the RSRS architecture concepts as explained above.

Summary of Architectural Evaluation. The Cortex project uses “Taster Databases” to find and filter out dangerous queries. If bad queries cause damage to a Taster Database, that query is not forwarded to the Master Database for actual processing. The Cortex system learns to distinguish bad queries from normal queries, and reconfigures the system to switch off damaged databases as well as regenerating new replacements. The Cortex system can be used by a self-healing monitor to protect Master Databases by managing the Taster Databases on behalf of the entire system.

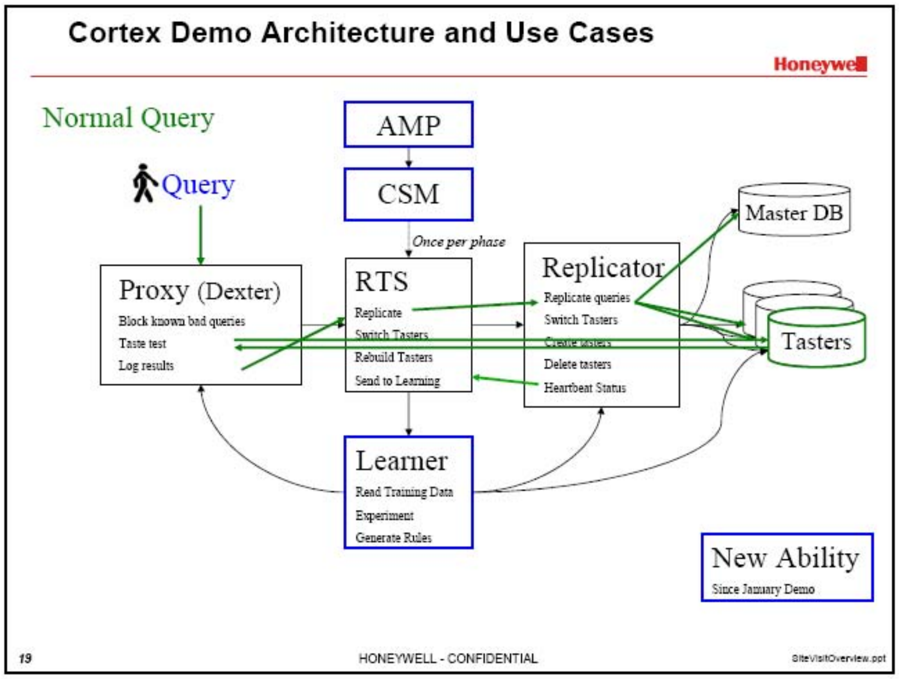


Figure 25 Structure of Cortex Demo Architecture

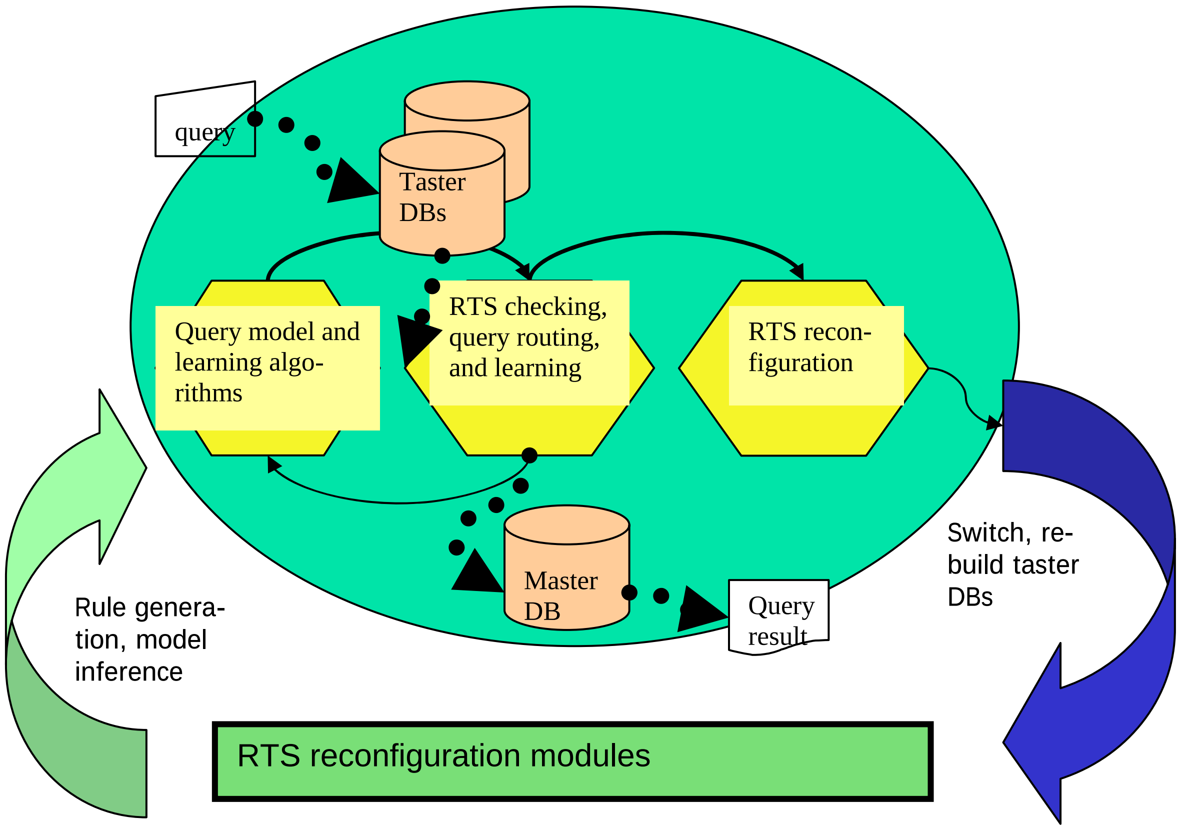

query
Taster DBs
Query model and learning algo-rithms
RTS checking, query routing, and learning
RTS recon-figuration
Master DB
Query result
Switch, re-build taster DBs
Rule genera-tion, model inference
RTS reconfiguration modules

Figure 26 Main MLA Features of Cortex

C.1.5 RSRS Architectural Comparison of Cognitive Projects

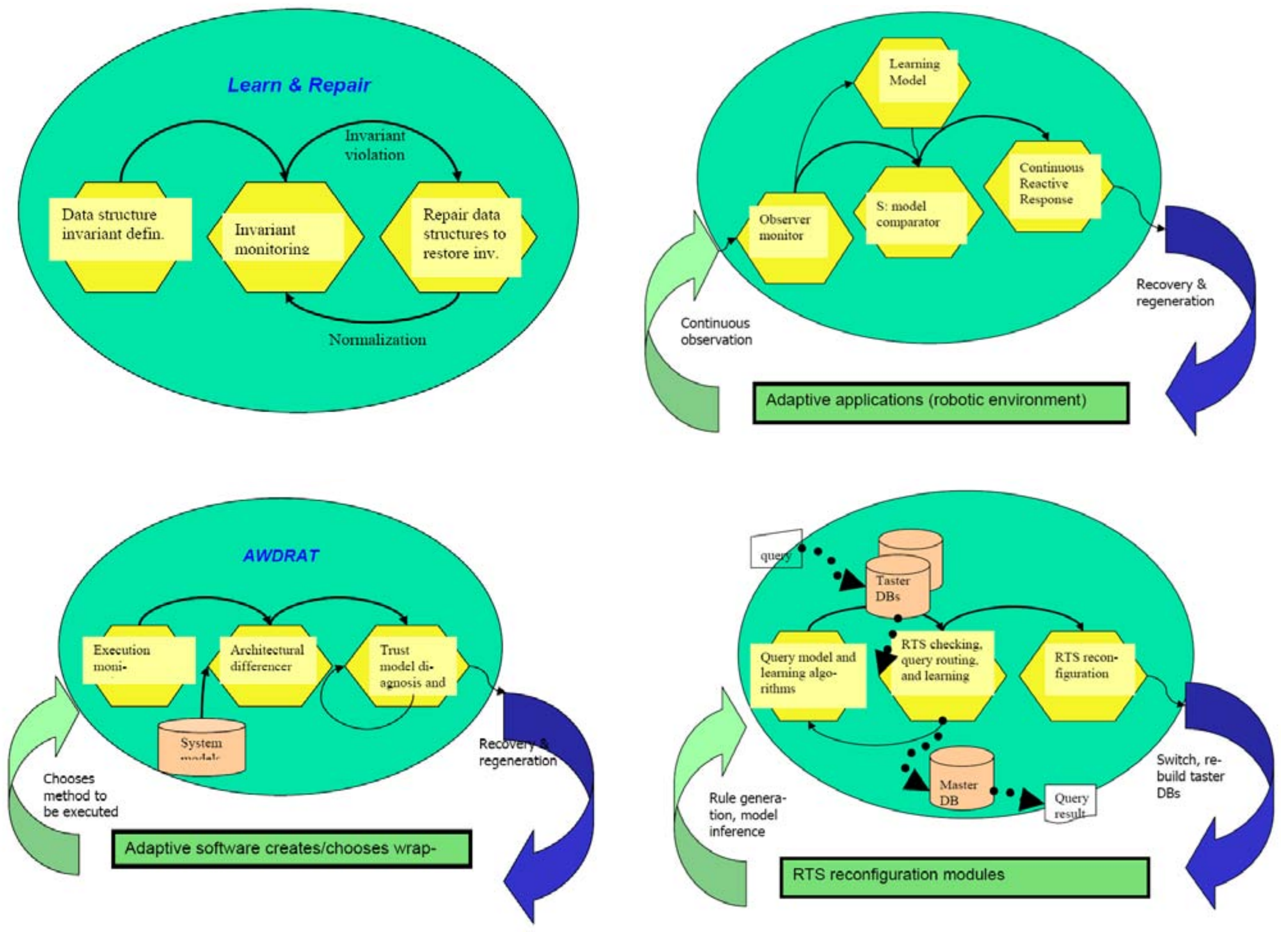

Learn & Repair
Invariant violation
Data structure invariant defin.
Invariant monitoring
Repair data structures to restore inv.
Normalization
Learning Model
Observer monitor
S: model comparator
Continuous Reactive Response
Continuous observation
Recovery & regeneration
Adaptive applications (robotic environment)
AWDRAT
Execution moni-
Architectural differencer
Trust model di-agnosis and
System models
Chooses method to be executed
Recovery & regeneration
Adaptive software creates/chooses wrap-
query
Taster DBs
Query model and learning algo-rithms
RTS checking, query routing, and learning
RTS recon-figuration
Master DB
Query result
Rule genera-tion, model inference
Switch, re-build taster DBs
RTS reconfiguration modules

Figure 27 RSRS Comparison of Cognitive Projects

At the program level, the Learn/Repair project is structurally different from the other three projects. The AWDRAT and Model-Based Executive projects are similar in structure and in their model-based approach. They differ in the application area chosen for the demonstration of the technology. The Cortex project is structurally similar to AWDRAT and Model-Based Execution, but focuses on the SQL query as the application area.

C.2 RSRS Architectural Evaluation of Diversity Projects

C.2.1 Genesis

Genesis (University of Virginia: John C. Knight, J.W. Davidson, D. Evans, A. Nguyen-Tuong; CMU: C. Wang)

Figure 28 illustrates the structure of the Genesis toolkit, taken from one of the slides of the Genesis presentation at 07/12/05 PI Meeting. Genesis is an integrated approach to inserting diversity into the physical representation of a program covering several abstraction layers when the program is being transformed. At the highest level, source code can be compiled into different object codes. At the next level, object codes can be translated into different executables at the link time. Then, executables can be loaded into different actual memory locations. Finally, dynamic diversity can be introduced at run-time, after the executable has been loaded.

Examples of specific techniques for creating effective diverse representations:

- Calling Sequence Diversity (CSD): protection against buffer-overflow and stack smashing attacks, by creating random memory layouts of branching addresses. CSD can be done at link or load time.
- Instruction Set Randomization (ISR): protection against machine code injection, by randomizing the machine instructions of the program being executed. ISR can be seen as a dynamic diversity method, since the randomized instruction set is interpreted and translated at run-time, although the randomization is typically done at code generation (compile) time.

Experiments: The Genesis project created diverse representations of the Apache web server. The representations are tested against known attacks and demonstrated their effectiveness. Experiments were conducted on the Strata VM and Red Hat Linux. The experiments show a very high percentage of success of a small number of diversification methods (e.g., CSD and ISR) when in "good" combination. In other words, the current generation of attack methods appears to be brittle when faced with simple diversification defense methods.

Figure 29 shows the RSRS architecture evaluation of Genesis, by transforming the structure shown in Figure 28 into the RSRS architecture concepts. We can see that the Genesis toolkit creates diverse variants and tests them against attacks. A matrix summarizing the test results shows which variants are immune when attacked by specific methods. Variants shown by the experiments as effective against known attacks (e.g., the combination CSD/ISR) can be used directly (as static and preventive defense), or loaded in when attacks happen (as dynamic defense).

Summary of Genesis Evaluation. The Genesis project generates program variants using techniques such as Calling Sequence Diversity and Instruction Set Randomization. The program variants can be tested and shown to be resistant to specific attacks. Some of the variants may also be immune to new attacks, for example, due to Instruction Set Randomization, which is difficult to guess. Their tool may be used by a system-level self-regenerative monitor to replace vulnerable programs or components either before or after an attack has happened.

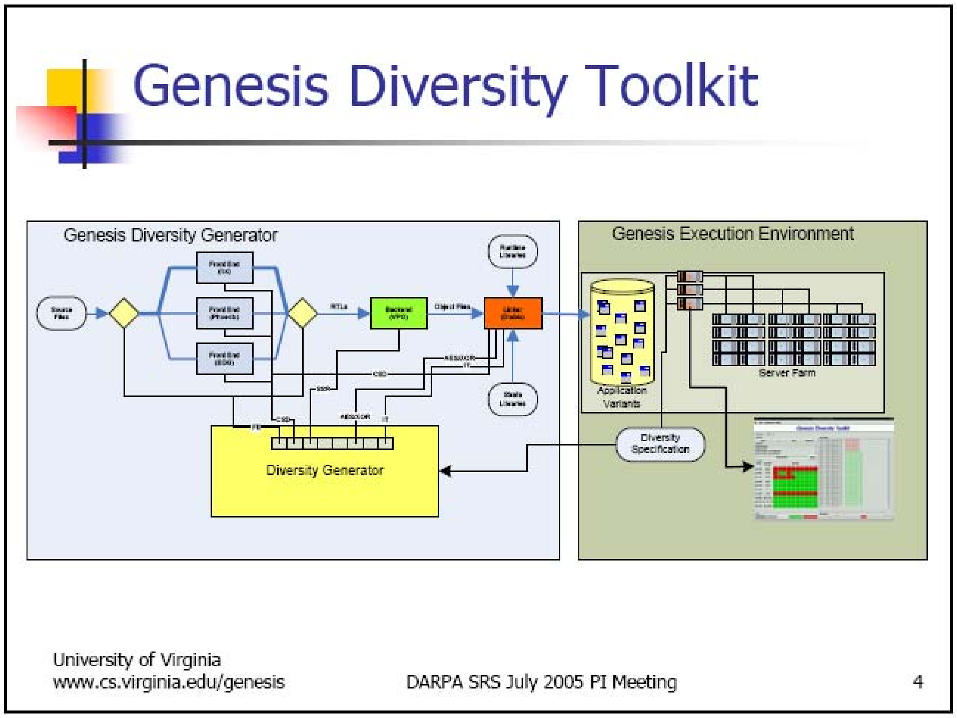

Genesis Diversity Toolkit
Genesis Diversity Generator
Genesis Execution Environment
Diversity Generator
Application Variants
Server Farm
Diversity Specification
University of Virginia
www.cs.virginia.edu/genesis
DARPA SRS July 2005 PI Meeting
4

Figure 28 Structure of Genesis Toolkit

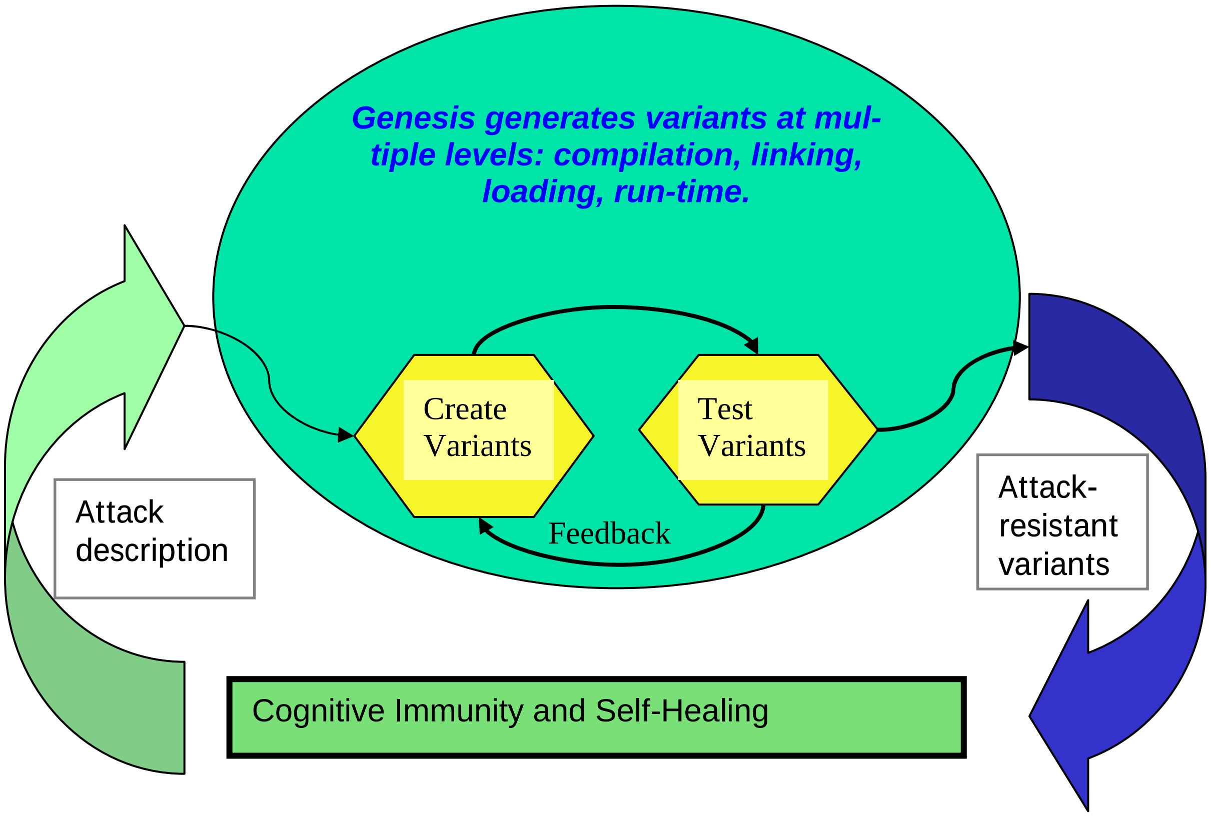

Genesis generates variants at multiple levels: compilation, linking, loading, run-time.
Create Variants
Test Variants
Feedback
Attack description
Attack-resistant variants
Cognitive Immunity and Self-Healing

Figure 29 Main MLA Features of Genesis

C.2.2 DAWSON – Synthetic Diversity for Intrusion Tolerance

DAWSON (Global Infotek: James Just)

Figure 30 illustrates the structure of Dawson toolkit, taken from one of the slides of the Dawson presentation at 07/12/05 PI Meeting. Dawson is a project similar in overall structure to Genesis, but more focused on the platform (Microsoft Windows binaries) and diversification techniques, primarily memory address randomization at linking and loading times. The project addresses memory-related vulnerabilities at two levels: exploit level (where the vulnerability is supposed to be) and the payload level (where the attack code is).

Examples of specific techniques for creating effective diverse representations:

- Transformation of absolute addresses, including DLL names, address allocation order, etc.
- Randomization of relative addresses.
- Unique randomizations at the load/start time of each execution by using pseudo random number generators.

Figure 31 shows the RSRS architecture evaluation of Dawson, by transforming the structure shown in Figure 30 into the RSRS architecture concepts. Similar to Genesis, we can see that the Dawson toolkit creates diverse variants using primarily memory address and name transformation techniques.

Summary of Self-Regenerative Functionality in DAWSON:

- Tripwires generating alerts of probable attacks from randomized applications
- Randomization transforms/key generation
- Automated re-randomization of applications at start-up
- Recovery, Restart, Re-randomization

Self-regenerative functionality includes the ability to alert other mechanisms of probable attacks. When applications are randomized, e.g. stack-based structures moved to other places, tripwires are placed in the conventional locations for such structures. For example, if the stack is padded with additional pages, the pages originally used by the un-randomized application are marked unreadable. Any attempted access to the original locations will generate alerts from tripwires inserted as part of randomization.

Randomly generated keys govern what transformations to apply when a system starts up. This can have some self-regenerative effect, in that programs that were vulnerable/incapacitated in a prior state can become invulnerable/capable in a later state. This self-regenerative effect is different from mere elimination of vulnerabilities or hardening of a program that also results from randomization. These hardening effects do not become self regenerative until the overall system takes advantage of the fact that a population of hardened programs or hosts survived an attack (or failure) and make use of this fact to regenerate those parts of the larger system that succumbed to attack.

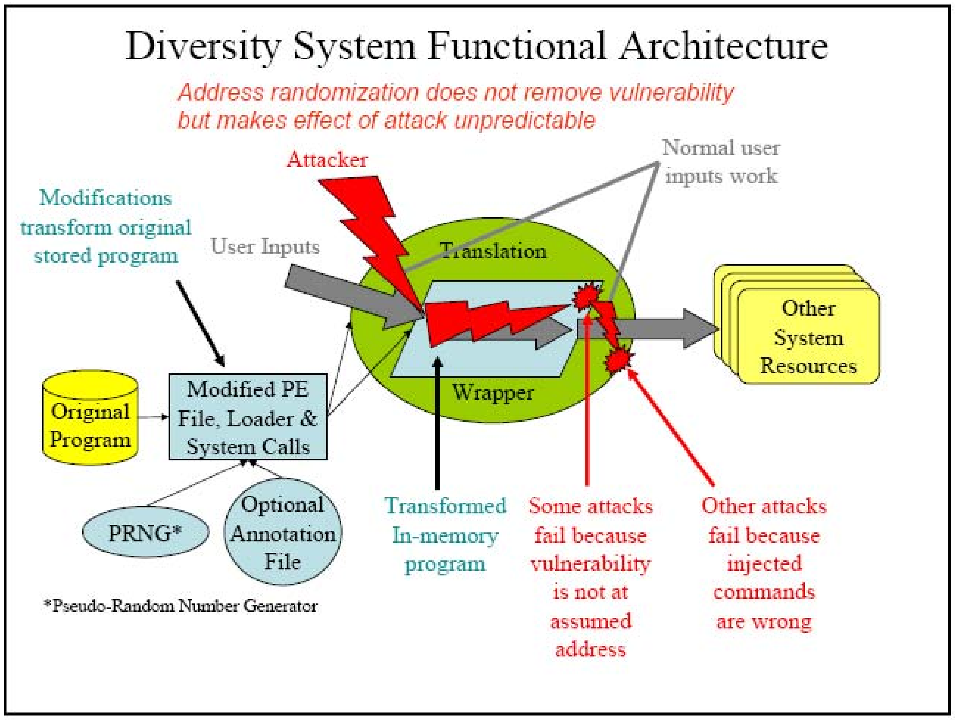

Diversity System Functional Architecture
Address randomization does not remove vulnerability but makes effect of attack unpredictable
Attacker
Normal user inputs work
Modifications transform original stored program
User Inputs
Translation
Wrapper
Other System Resources
Original Program
Modified PE File, Loader & System Calls
PRNG*
Optional Annotation File
Transformed In-memory program
Some attacks fail because vulnerability is not at assumed address
Other attacks fail because injected commands are wrong
*Pseudo-Random Number Generator

Figure 30 Structure of Dawson Toolkit

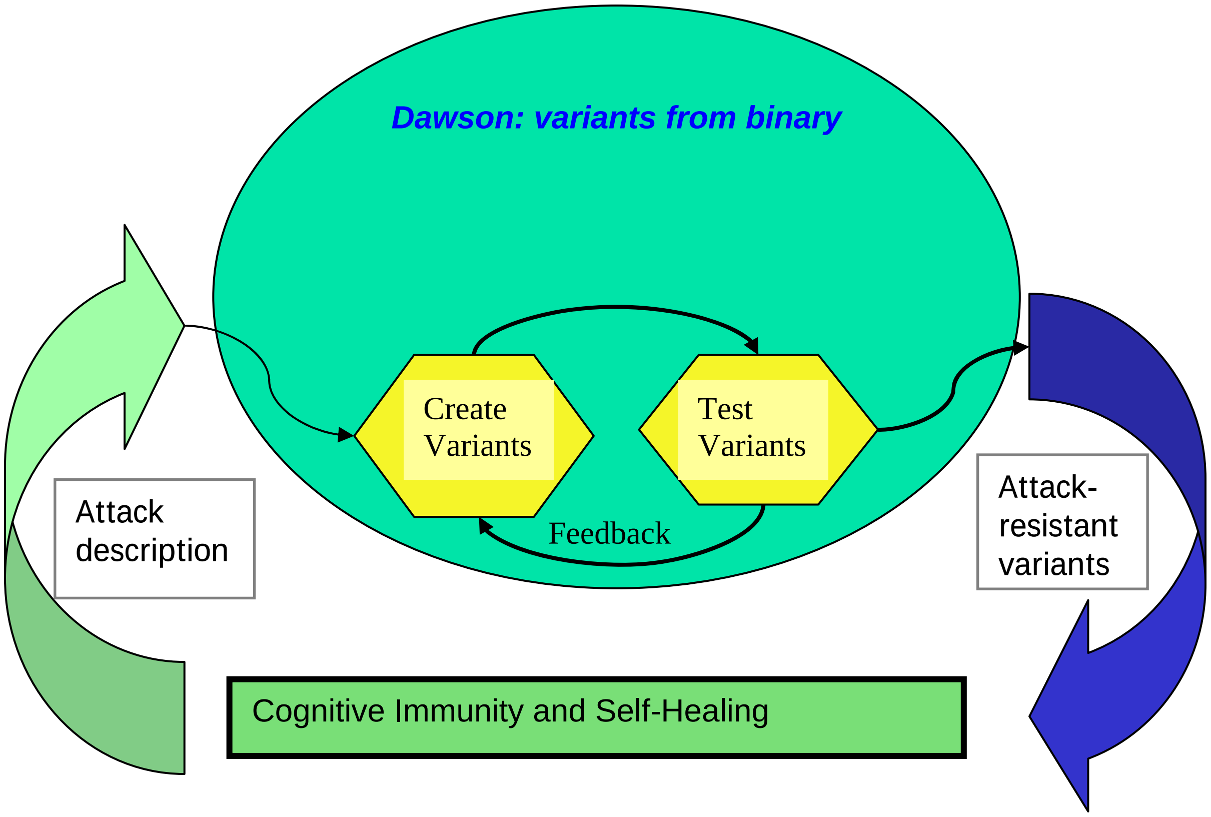

Dawson: variants from binary
Create Variants
Test Variants
Feedback
Attack description
Attack-resistant variants
Cognitive Immunity and Self-Healing

Figure 31 Main MLA Features of Dawson

When programs fail, they can be killed and/or restarted as necessary as a countermeasure. Programs may fail either because of attack or some instability introduced by our randomizing transforms. The additional levels of error handling introduced by the Dawson transformations can make applications more robust in the face of errors. When errors are detected, applications can be killed and restarted as part of this error handling. With such mechanisms there is the potential to turn over decision making on whether and when to restart to external components (mechanisms in the broader architecture.) Hooks have not yet been implemented to enable this.

The following tools are provided by DAWSON:

- Application hooking + native service randomize user level runtime structures
- Kernel mode driver for low level randomization management
- Routines for dynamic randomization of specific runtime structures

The project's functionality is concerned with specific interfaces and control mechanisms within its specific software infrastructure. The DAWSON software infrastructure is exclusively Windows based, Windows being the most pervasive and insufficiently diverse mono-culture the problems of which DAWSON is designed to mitigate.

Randomization at the user level is achieved by a user mode hooking approach. In this approach branch tables, or in some cases addresses within the code are patched dynamically to gain control at critical points. At this time injected code can run and perform specific randomization functions before returning control at appropriate points. Much of this injected code makes use of native API's exposed (but not documented) on the NTDLL interface. In the future, the randomizing transforms may be controlled by an external properties file, entries in the registry, or some distributable service that delivers these parameters the host in a coordinated manner.

The following specific randomizations are provided by DAWSON:

- Stack randomization – where the run-time stack is positioned
- Heap randomization – positions of blocks in the run-time heap
- DLL randomization – where DLLs get positioned in process memory
- IAT table randomization – shuffles entries in the Import Address Table

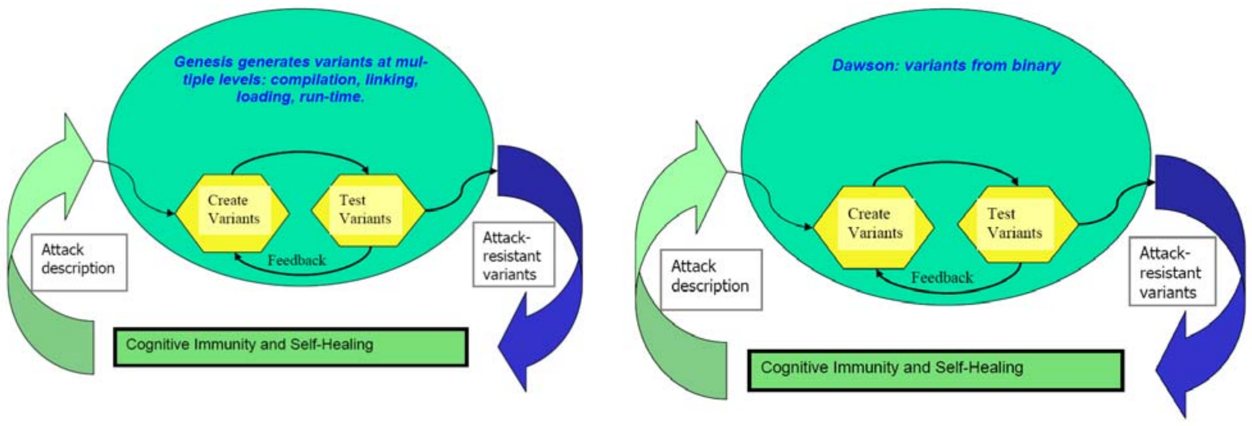
The project is currently working on a replacement implementation of the Windows CreateProcess call that gives finer grained control over process creation at the kernel level. This gives more control that can be achieved with hooks to individual calls. For example, arbitrary process creations can be detected and the loading of primitive structures into memory at boot time can be controlled. The plan is for CreateProcess to work in conjunction with a small kernel driver that does things that can only be done in kernel mode and leave most of the randomization in user mode. A kernel mode driver can detect process creation earlier and apply randomizations there.

The project has worked with static transformations that randomize individual PE files (EXEs and DLLs) generating a new PE file before the PE file is loaded into memory. These transformations take a PE file and generate a new PE file as a result. This mechanism will be supported with an interface consistent with the dynamic transformation interface; the difference being that the transformation will be applied to the file before the file is loaded into memory. The approach has an advantage that some parameters interpreted by the loader may be manipulated before the loader interprets them, rather that after as in the case of our dynamic transformations. The primary disadvantage is that maintaining the integrity of the PE file can be complex for some desirable transforms. Success with dynamic transformation of run-time structures in memory has caused the project to de-emphasize these static transformations of PE files.

Another potential interface is the set of routines that implement the transforms themselves. A small library of routines to perform specific transformations has been accumulated. Right now these routines are embedded in the code of a hooking application. However, these routines could be pluggable, conforming to a standard API. This could provide a potential integration point with self-regeneration control software.

Summary of DAWSON Evaluation. The DAWSON project generates program variants for the Windows environment using techniques such as variable location (stack/heap) randomization and address (DLL/IAT) randomization. The program variants can be tested and shown to be resistant to specific attacks. Their tool may be used by a system-level self-regenerative monitor to replace vulnerable programs or components either before or after an attack has happened.

C.2.3 RSRS Architectural Comparison of Diversity Projects



Genesis generates variants at multiple levels: compilation, linking, loading, run-time.
Create Variants
Test Variants
Feedback
Attack description
Attack-resistant variants
Cognitive Immunity and Self-Healing
Dawson: variants from binary
Create Variants
Test Variants
Feedback
Attack description
Attack-resistant variants
Cognitive Immunity and Self-Healing

Figure 32 RSRS Comparison of Diversity Projects

The two Diversity projects are structurally similar, but have different deliverables (different operating systems and system environment assumptions).

C.3 RSRS Architectural Evaluation of Redundancy Projects

C.3.1 Scalability, Accountability and Instant Information Access for Network-Centric Warfare

SAIIA (Johns Hopkins University: Yair Amir; Purdue University: Cristina Nita-Rotaru)

The SAIIA project is studying scalable and efficient Byzantine fault tolerance for systems that span a wide-area network environment. As part of the project, the Steward architecture, depicted in Figure 33, was developed. This figure was taken from one of the slides of the SAIIA presentation at 07/12/05 PI Meeting. Steward makes use of expensive Byzantine fault-tolerant object replication protocols only within local sites while employing more efficient non-Byzantine fault-tolerant protocols across sites. The SAIIA project is also exploring update accountability in which the effects of bad updates can be traced in a post-mortem analysis once the bad update is detected.

SAIIA Project Summary: The project is studying 3 innovative ideas:

- Scalable wide-area intrusion tolerance: expensive Byzantine fault-tolerant replication and threshold cryptography used within a site, more efficient non-Byzantine fault-tolerant replication across wide area
- Update accountability: dependency tracking used to mark corrupt and possibly corrupt data after a bad update is detected – this requires some intrusion detection mechanism, which is not discussed
- Instant information access: fast update propagation using commutative update semantics that avoids the need to globally order updates

The current focus of the project is scalable wide-area intrusion tolerance, for which the Steward architecture was developed.

RSRS Architectural Analysis: Within the Steward wide-area intrusion-tolerant architecture, there is monitoring and regenerative actuation:

- Monitoring: The standby servers have monitors (see Figure 33) that are responsible for verifying that the representative server executes the wide-area protocol correctly.
- Regenerative Actuation: Detection of incorrect operation of the representative server results in removal of the current representative and election of a new representative.

Update accountability can also be modeled as an MLA loop:

- Monitoring: Intrusion detection mechanisms monitor and detect bad updates.
- Learning-Based Diagnosis: Optional.
- Regenerative Actuation: Update accountability mechanisms trace corrupted data based on bad update identification, thereby enabling recovery mechanisms to be invoked.

Several technologies being developed in the SAIIA project could be used as stand-alone tools to provide support for self-regenerative systems and/or applications. These include protocols for wide-area object replication and threshold cryptography.

Summary of SAIIA Evaluation. The Steward wide-area object replication work is the most relevant to the SRS program. It provides a useful technology to support intrusion-tolerant systems that are deployed across wide-area network environments. Currently, the project has been primarily focused on achieving performance goals, as called for by the SRS program, rather than investigating self regeneration within a replicated environment. There is a significant future opportunity to enhance object replication mechanisms with internal self regeneration so that they can not only support larger self-regenerative systems but can also provide inherently self-regenerative replicated objects.

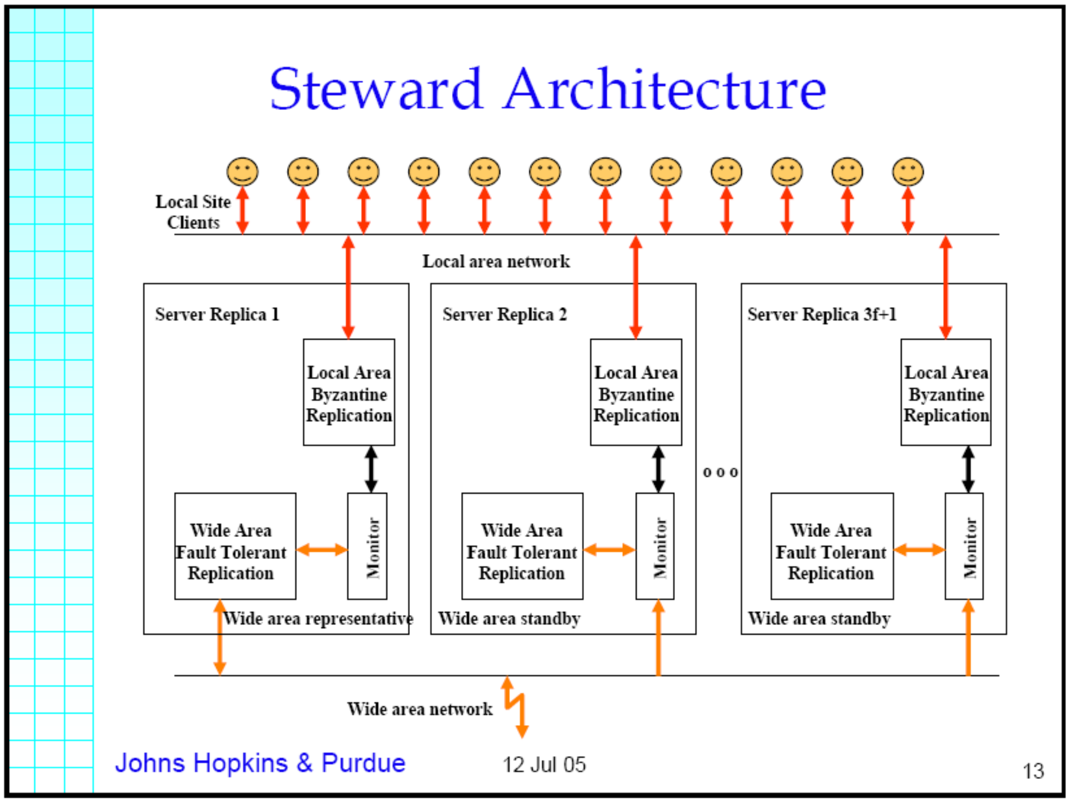


Figure 33 Structure of Steward Architecture (J.Hopkins)

C.3.2 Increasing Intrusion Tolerance via Scalable Redundancy

IITSR (CMU: Michael K. Reiter, Greg Ganger, Priya Narasimhan, Anastassia Ailamaki, Chuck Cranor)

The technologies explored in the IITSR project provide intrusion-tolerant data storage that can be used in support of self-regenerative systems. For example, IITSR technologies could be used to securely store the knowledge base maintained by cognitive area projects. Figure 34 illustrates the structure of the update operation in a Byzantine-fault-tolerant query/update system, taken from one of the slides of the IITSR presentation at 07/12/05 PI Meeting. The Byzantine-fault-tolerant query/update system is one of the technologies being investigated in IITSR.

IITSR Project Summary: The project considers both low-cost data replication and higher-cost object replication, as well as specialized protocols for important object types that judiciously mix data/object replication to achieve good performance. Additional performance improvements are expected through the use of techniques such as shifting load from servers to clients, not requiring that invocations be processed on *all* servers, using versioning to avoid proactive update ordering, etc.

RSRS Architectural Analysis: IITSR replication management protocols contain simple rules that may be modeled as MLA loop. The server proceeds with version-based updates, and clients are responsible for detecting and resolving update conflicts.

- Monitoring: The replication mechanisms have various "bad update" detection capabilities, e.g. failure to reach agreement when "agreed write" is employed and failure to verify data in verifiable read protocol.
- Learning-Based Diagnosis: The data replication mechanisms in IITSR are compatible with several existing replication-based diagnosis algorithms.[3] These algorithms could be used with IITSR to diagnose faulty and compromised data servers.
- Regenerative Actuation: Bad updates that are detected at the time of a subsequent read could be recovered from either by regenerating correct data (possible in some cases) or reverting to a previous version (when regeneration is not possible).

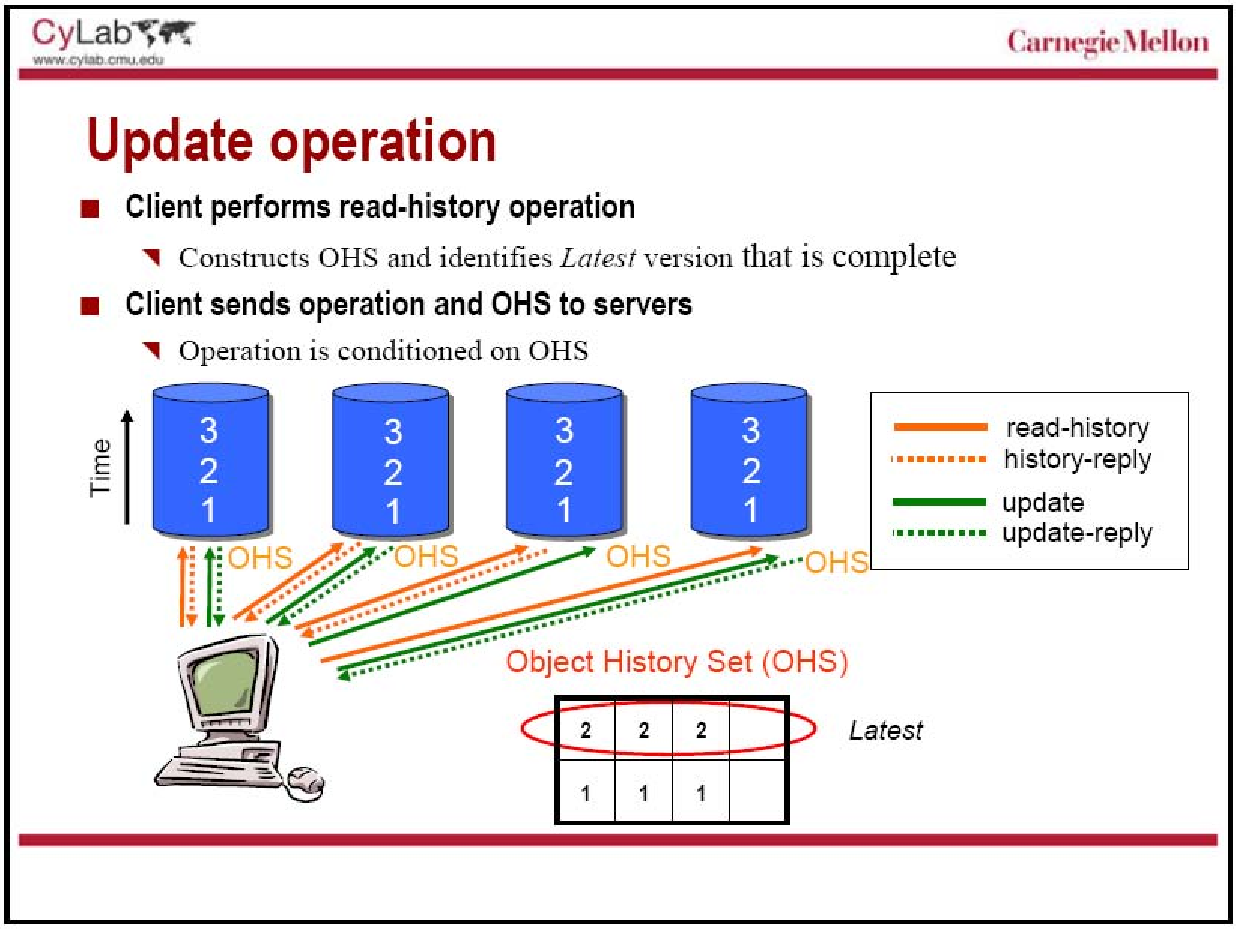
A number of technologies being developed in the IITSR project could be used as stand-alone tools to provide support for self-regenerative systems and/or applications. These include server-side versioning, Byzantine protocols for read/write data, linearizable data access mechanisms, and Byzantine protocols for query/update objects.

Summary of IITSR Evaluation. Currently, the IITSR project focuses primarily on providing supporting technologies for SRS. An unexplored but extremely promising extension would be to provide a self-contained self-regenerative data store, i.e. to incorporate aspects of self regeneration inside the data store itself. Such a self-regenerative data store is indicated in Figure 4, where the MLA loop characterizing self regeneration is present inside the data/object store. Techniques for diagnosis, recovery, reconfiguration, and adaptation of Byzantine-fault-tolerant data access technologies have been studied in several projects.[3,4] These techniques are well suited for integration with the types of technologies developed within IITSR. Successful integration would result in the self-contained self-regenerative data store envisioned in Figure 4. Self-contained self-regenerative components such as these could form building blocks from which larger self-regenerative systems could be constructed. Development of important self-regenerative components such as a general-purpose data store would therefore constitute an extremely valuable contribution of the SRS program.

[3] Alvisi, Malkhi, Pierce, and Reiter, "Fault Detection in Byzantine Quorum Systems," *IEEE Transactions on Parallel and Distributed Systems,* Sept. 2001.

Kong, Subbiah, Ahamad, and Blough, "A Reconfigurable Byzantine Quorum Approach for the Agile Store," *Proceedings of the 2003 Symposium on Reliable Distributed Systems*.

[4] Kong, Manohar, Subbiah, Ahamad, and Blough, "Agile Store: Experience with Quorum-Based Replication Techniques for Adaptive Byzantine Fault Tolerance," *Proceedings of the 2005 Symposium on Reliable Distributed Systems,* to appear.



CyLab www.cylab.cmu.edu

Carnegie Mellon

Update operation

- Client performs read-history operation
  - Constructs OHS and identifies *Latest* version that is complete
- Client sends operation and OHS to servers
  - Operation is conditioned on OHS

Figure 34 Example Structure of CMU Project

C.3.3 QuickSilver

QuickSilver (Cornell: Ken Birman, Paul Francis, Johannes Gehrke, Robbert van Renesse; Raytheon: Jay Lala, Tom Bracewell)

The Quicksilver project is developing several scalable reliable communications technologies for use within self-regenerative systems. For example, Quicksilver and its Cayuga event processing system can be used to disseminate and process important events, such as failures and intrusions that are detected at various levels by different monitors and sensors in the system. Figure 35 illustrates the structure of the Cayuga system, taken from one of the slides of the QuickSilver presentation at the 07/12/05 PI Meeting.

Quicksilver Project Summary: Quicksilver is developing three primary technologies: the Cayuga event processing system, the SlingShot scalable reliable multicast service, and the Quicksilver scalable publish-subscribe infrastructure. The Cayuga system offers scalable XML event filtering. SlingShot uses newly developed techniques for scalable reliable multicast to provide scalable clustered services via a Web Service paradigm. The packet recovery mechanism for SlingShot is depicted in Figure 36, which is taken from one of the slides of the

QuickSilver presentation at the 07/12/05 PI Meeting. Quicksilver pub/sub provides a scalable reliable infrastructure for publish/subscribe applications.

RSRS Architectural Analysis: As with the other GSR projects, Quicksilver primarily provides technologies to support self-regenerative systems. Most of the developed technologies provide inherent robustness to some level of intrusion but are not explicitly self-regenerative. For example, the SlingShot scalable reliable multicast technology uses gossip-based protocols that are robust to a limited number of misbehaving participants, however it does not attempt to identify and remove misbehaving participants in order to fully heal the system.

Summary of QuickSilver Evaluation. The Quicksilver technologies fit well within the RSRS architecture, as is depicted in Figure 4. Important events, such as failures and intrusions that are detected at various levels by different monitors and sensors in the system, must be disseminated and processed using a system such as Cayuga. Applications communicate internally and externally using GSR communications mechanisms such as reliable multicast (SlingShot) and publish/subscribe (Quicksilver pub/sub). In order to support the RSRS architecture, all of the communication and event processing must be done in a scalable and reliable manner, using technologies such as those developed in the Quicksilver project. It should also be emphasized that QuickSilver is the *only GSR project focusing on scalable reliable communications services.* The other two GSR projects (SAIIA and IITSR) are focused on data and/or object replication mechanisms, rather than communication.

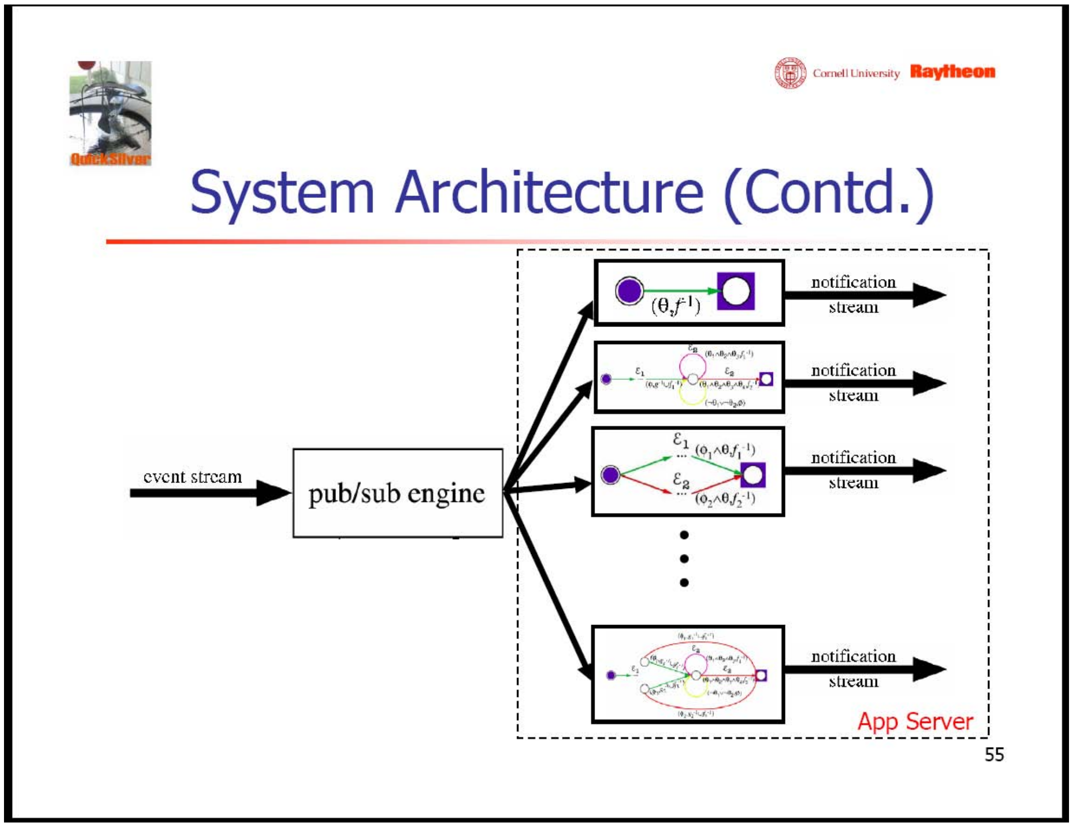
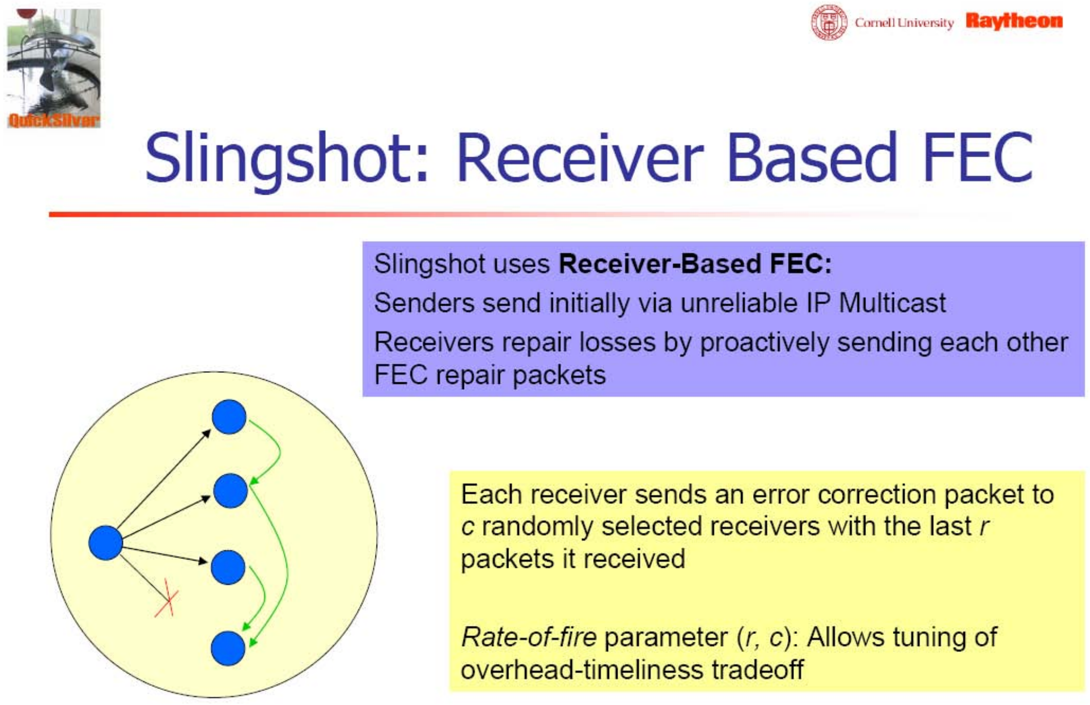


Figure 35 Event-System Architecture in Cayuga



Cornell University Raytheon

QuickSilver

Slingshot: Receiver Based FEC

Slingshot uses **Receiver-Based FEC:**

Senders send initially via unreliable IP Multicast

Receivers repair losses by proactively sending each other FEC repair packets

Each receiver sends an error correction packet to *c* randomly selected receivers with the last *r* packets it received

Rate-of-fire parameter (*r*, *c*): Allows tuning of overhead-timeliness tradeoff

Figure 36 Packet Recovery Mechanism in SlingShot

C.3.4 RSRS Architectural Comparison of Redundancy Projects

The three GSR projects are quite different in focus. SAIIA deals primarily with general object replication over wide-area networks. IITSR focuses on data replication with some consideration of flat objects. QuickSilver considers scalable and reliable communication. None of the GSR projects has considered internal self-regenerative aspects in detail; instead, they have focused on obtaining the best performance while providing functionalities of use to a larger self-regenerative system. Due to the redundant structures they employ, all of the projects have some capability to tolerate intrusions and attacks so long as they do not affect too many modules.

C.4 RSRS Architectural Evaluation of Insider Projects

C.4.1 Detecting and Preventing Misuse of Privilege

PMOP (Teknowledge: Bob Balzer; MIT: Howie Shrobe)

Figure 37 illustrates the structure of PMOP architecture, taken from one of the slides of the PMOP presentation at 07/12/05 PI Meeting.

Summary: the PMOP Architecture can be modeled as the ML part of the MLA loop. The action part is implicit in the architecture, with potentially explicit regeneration action needed.

- Monitoring: *Operator Behavior Monitor* - a component that mediates the communication between a legacy system's GUI and the system itself to extract the application level commands or directives initiated by the user/operator through that GUI so that they can be screened for harmful effects before being processed by the legacy system.
- Learning-Based Diagnosis:
    - *Matching Operator Behavior against Role-Based Plans* - a component that compares operator behavior traces to behavior traces from operator and attack plans.
    - *Operational System Model* - an operational system model for a legacy application - initially constructed from propositional rules – from which both the predicted state of the system, and the likelihood of harm resulting from the change of state can be predicted.
    - *Malicious Behavior Detector* - a suspicious behavior detector that differentiates between accidents and malicious behavior by inferring user goals from the observed harmful behavior, recent historical behaviors, and the set of plans consistent with the larger behavior context.
- Regenerative Actuation: The mechanism to distinguish between benign actions and harmful actions, followed by authorization or disallowing of actions, is in the architecture. However, if the operational system model needs to be changed (e.g., new information that changes some benign actions to harmful), the Harm Assessment module will need to be updated.

Figure 38shows the RSRS architecture evaluation of the PMOP project, by translating the feedback structure shown in Figure 37 into the RSRS architecture concepts.

Summary of PMOP Evaluation. The PMOP project uses an operator behavior monitor to compare the expected actions (as defined by an Operational System Model) with the actual operations. If deviations are found, the Harm Assessment Module checks whether the extraordinary actions are dangerous. Dangerous actions go through Intent Assessment, which distinguishes malicious insider actions from operator errors. PMOP tools may be used to observe any applications for which a good set of models can be defined (Operational System Model, Harm Model, Intent Model).

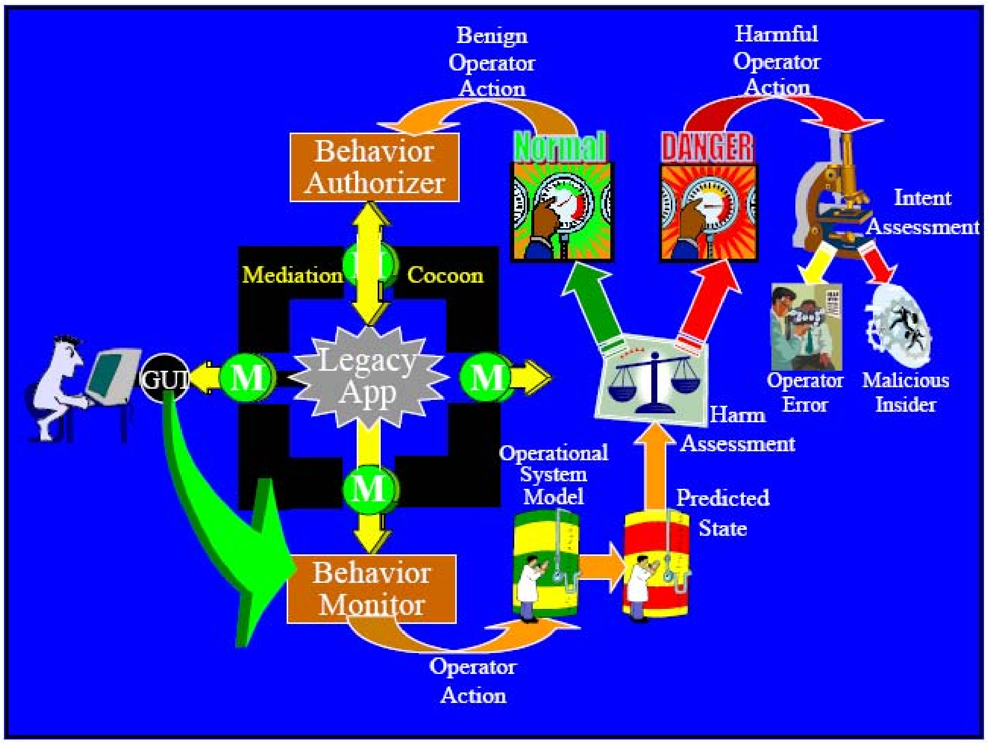
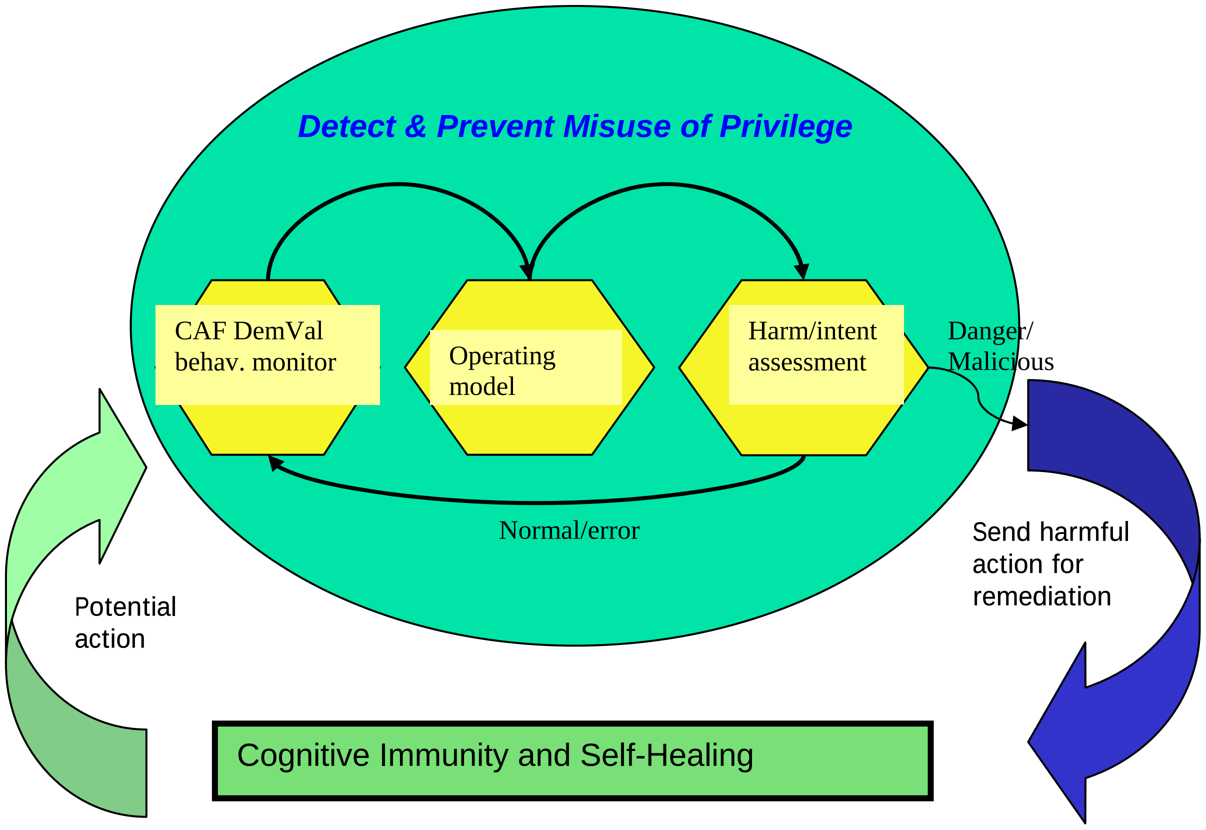


Figure 37 Structure of PMOP Architecture and Data Flow



Detect & Prevent Misuse of Privilege
CAF DemVal behav. monitor
Operating model
Harm/intent assessment
Danger/ Malicious
Normal/error
Potential action
Send harmful action for remediation
Cognitive Immunity and Self-Healing

Figure 38 RSRS Insider Project (PMOP)

C.4.2 Mitigating the Insider Threat using High-Dimensional Search and Monitoring

HDSM (Telcordia: Eric van den Berg; Rutgers: Raj Rajagopalan)

Figure 39 illustrates the structure of HDSM architecture, taken from one of the slides of the HDSM presentation at 07/12/05 PI Meeting.

Summary: the approach can be modeled as MLA loop.

- Monitoring:
    - A very large **sensor network** that collects data from numerous diverse sensors monitoring various system layers from physical to network layer to application layer and end-host sensors, making it very hard for any suspicious insider behavior to avoid triggering some sensor alerts.
    - A **network history repository**, which contains historical states of the system that are annotated with attack and precursor information. We will deliver a design document and prototype of a network state description language in terms of collected sensor data.
- Learning-Based Diagnosis:
    - A **high-dimensional search engine** operating on the network history repository, that is based on dimension reduction techniques such as Singular Value Decomposition (SVD).
    - A graph-based **insider threat modeling and analysis** tool, which identifies potential insider attack points and attack scenarios in a network by modeling the effort required to acquire knowledge of internal details, and provides the required "experience" for the pre-attack training phase.
- Regenerative Actuation:
    - A **response engine**, which performs an impact analysis of the potential attack on critical services and automatically synthesizes a response that minimizes collateral damage, thwarting the predicted attack within minutes.

Figure 40 shows the RSRS architecture evaluation of the HDSM project, by translating the structure shown in Figure 39 into the RSRS architecture concepts.

Summary of HDSM Evaluation. The HDSM project uses a large sensor network to collect behavioral information of operators. These data are stored in a network history repository, based on which a high-dimensional search engine will learn to distinguish proper actions from insider threats. An insider threat modeling and analysis tool builds models of insider knowledge acquisition that precedes attacks. A response engine performs impact analysis and synthesizes countermeasures that minimize potential damage. Their tools may be used to observe and detect insider threats provided appropriate sensors and models can be built and deployed for the operations being observed.

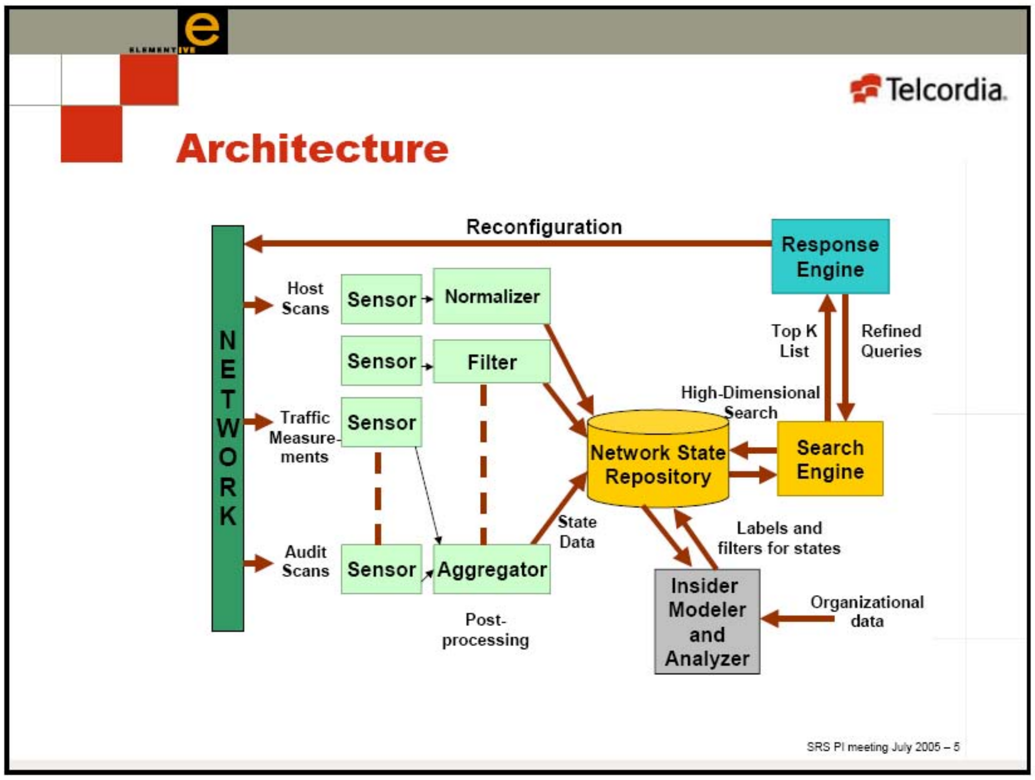


Figure 39 Structure of HDSM Architecture

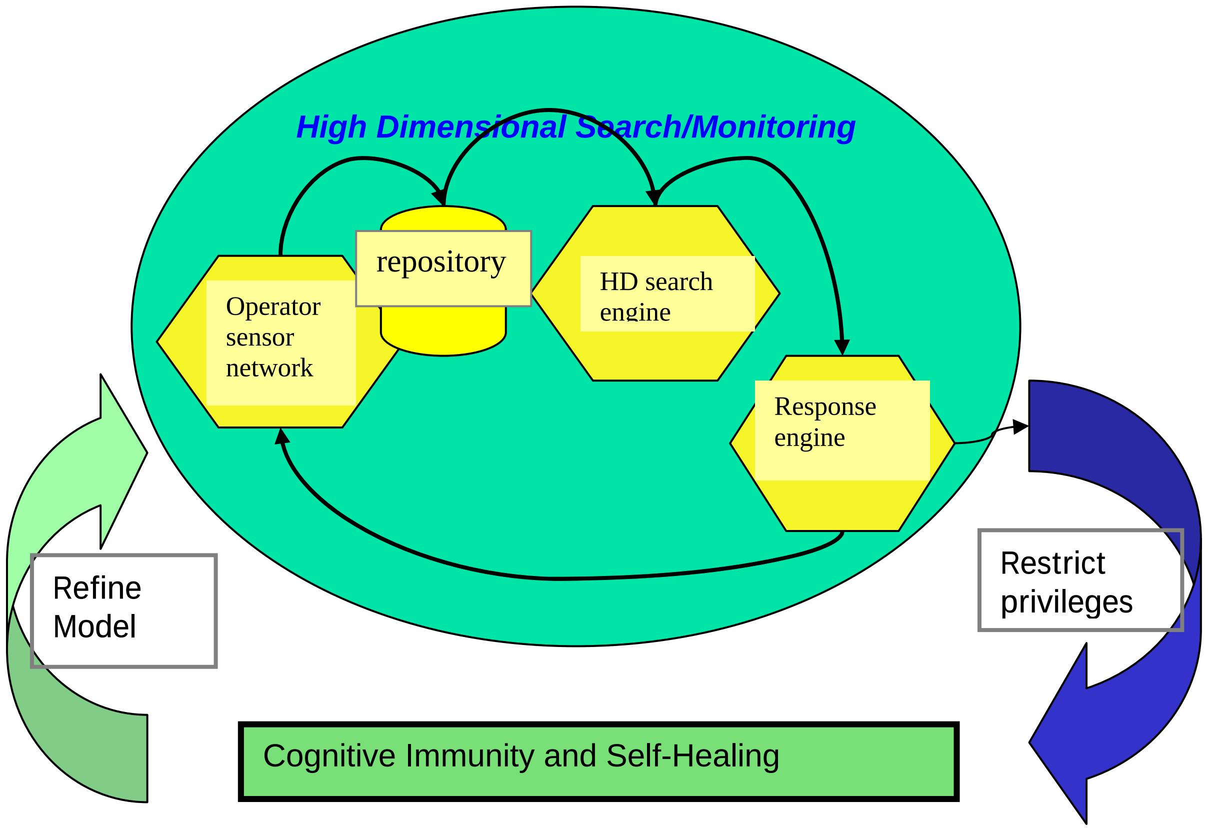


Figure 40 RSRS Insider Analysis (HDSM)

C.4.3 RSRS Architectural Comparison of Insider Projects

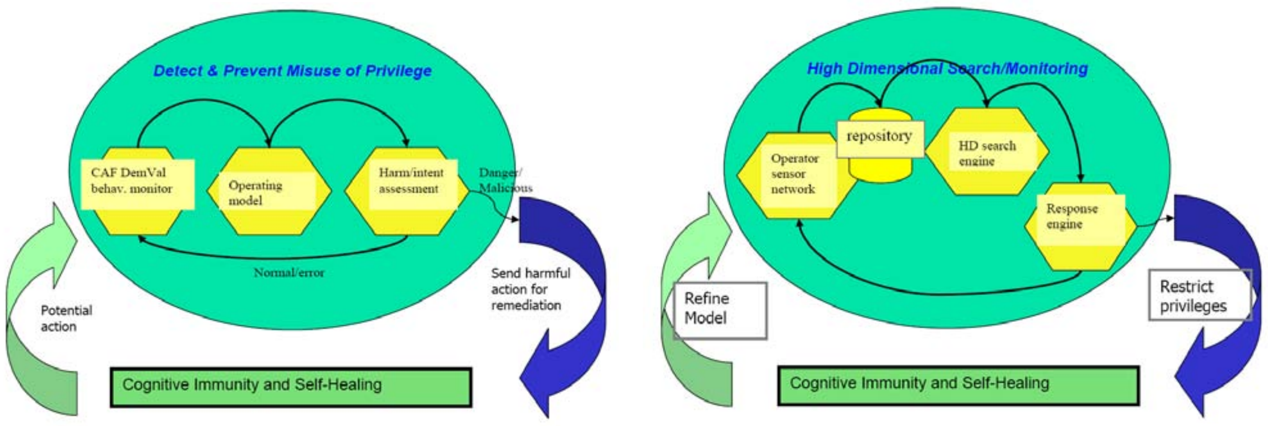

Detect & Prevent Misuse of Privilege
CAF DemVal behav. monitor
Operating model
Harm/intent assessment
Danger/ Malicious
Normal/error
Potential action
Send harmful action for remediation
Cognitive Immunity and Self-Healing
High Dimensional Search/Monitoring
repository
HD search engine
Operator sensor network
Response engine
Refine Model
Restrict privileges
Cognitive Immunity and Self-Healing

Figure 41 RSRS Comparison of Insider Projects

The two Insider projects (not counting Asbestos) are structurally similar, with more emphasis on data mining in the HDSM project.

D. Summary of RSRS Analysis and Evaluation

In this study, we have applied the RSRS architecture to analyze and evaluate the current SRS projects, divided into four areas: Cognitive, Diversity, Redundancy, and Insider. Our main effort is on using the MLA (monitor-learning-actuator) feedback loop to capture the self-regenerative aspects of the projects. Our second effort is evaluating the combination of diversity properties with self-regenerative properties in the achievement of quantitative SRS program goals.

At the top level, the study shows that the MLA loop is an effective modeling tool for the four projects in the Cognitive area, the two projects in the Diversity area, and two projects in the Insider area (except for Asbestos, which provides isolation instead of threat identification). The three projects in the Redundancy area use replication techniques to provide diversity regardless of MLA feedback.

At the next level, we have used MLA loop (when applicable) and diversity techniques to analyze the architecture and main functionality goals of each project. In this report (Part 1 of the study), we have found each project to provide unique and complementary functionality, if completed as proposed. A scenario for the combination of the SRS project results into a useful system is the subject of the second report.

Acknowledgements

This report contains contributions from several SRS performers, including Michael Ernst, Mark Cornwell, Yair Amir, and Tom Bracewell. The report also benefited from comments by Chris Geib and John Knight.